Vysoká škola báňská – Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra informatiky

Vykonávání XPath dotazů na XML datech validních k existujícímu XML Schema

XPath processing on an XML data with the existing XML Schema

2014 Bc. Marek Lakoštík

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Marek Lakoštík**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Vykonávání XPath dotazů na XML datech validních k existujícímu XML Schema
XPath Processing on an XML Data with the Existing XML Schema

Zásady pro vypracování:

Holistické algoritmy, které slouží k vykonávání XPath dotazů, jsou navrženy tak, aby pracovaly s nějakým indexem (např. invertovaný seznam). Dají se ovšem použít i v případě, že XML dokument čteme sekvenčně a žádné předzpracování XML dokumentu není možné. V takovém případě může být užitečná znalost odpovídajícího XML schema. Tato práce se bude zabývat optimalizací holistického algoritmu s využitím XML schema při sekvenčním zpracování dokumentu.

Práce bude probíhat v následujících krocích:
1. Seznámení se s algoritmem GTPStack a optimalitou holistických algoritmů.
2. Studium algoritmů zabývajících se vykonáním XPath dotazů s použitím XML schema.
3. Návrh a implementace modulu, jenž bude funguvat následujícím způsobem:
a) zpracuje XML schema k daným XML dokumentům do příslušného grafu,
b) ponechá příslušnou kostru grafu odpovídající dotazu, potřebné informace předá GTPStack algoritmu,
c) začne zpracovávat vstupní XML dokument tak, aby jej bylo možné číst GTPStack algoritmem.
4. Důkladné otestování aplikace.

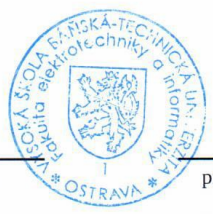
Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Radim Bača, Ph.D.**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2014



doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne.
Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

V Ostrave 1.5.2014 .....

Poďakovanie

Chcel by som poďakovať všetkým, ktorí mi akýmkoľvek spôsobom pomohli pri spracovaní tejto diplomovej práce. Moje poďakovanie patrí najmä vedúcemu práce Ing. Radimovi Bačovi, Ph.D. za vedenie, úsilie, čas a poskytnutie cenných rád, bez ktorých by táto práca nevznikla.

Abstrakt

XML je značkovací jazyk, který se v současnosti stále více používá pro uchovávání dat v oblasti informačních technologií. Podobně jako se dá vyhledávat v relačních databázových systémech pomocí jazyka SQL, dá se vyhledávat i v XML databázových systémech. Nejpoužívanějším XML dotazovacím jazykem je XPath, jehož dotazy jsou zpracovávány kromě jiných i holistickými algoritmy. Ty jsou navrženy tak, aby pracovaly s určitou datovou strukturou, která obsahuje předzpracovány uzly XML dokumentu tak, aby bylo možné k nim efektivně přistupovat. Hlavním cílem této práce je otestovat funkčnost holistických algoritmů v případě, že neexistuje žádné předzpracování XML dokumentu, a XML dokument je zpracováván na vstupu sekvenčně. Neodmyslitelnou součástí činnosti holistických algoritmů je práce s proudy uzlů XML dokumentu. Efektivita práce těchto algoritmů je závislá i na velikosti těchto vstupních proudů. V této práci se zabýváme i redukcí uzlů ve vstupních proudech algoritmu za využití XML Schema validního k zpracovávanému XML dokumentu. Zmiňovaná redukce pracuje na základě prefixové a postfixové optimalizace větveného dotazu. Větvený dotaz představuje jeden z nejčastěji řešených typů dotazů, které řeší holistické algoritmy.

Klíčová slova

XPath, XML, XML strom, XML Schema, DTD, validace, orientovaný graf, pre-order procházení grafu, větvený dotaz, holistické algoritmy, PathStack, TwigStack, GTPStack

Abstract

XML is a mark-up language, which is increasingly used for storing data in the field of information technology. Like it is possible to search in relational databases with help of SQL language, it is also possible to search in XML databases. XPath is the most commonly used XML query language, which queries are processed, besides others, by holistic algorithms. These are designed to work with a particular data structure that contains the XML document nodes pre-processed so that these can be effectively treated. The main goal of this work is to test functionality of holistic algorithms when there is not any pre-processing of XML document and XML document is being processed sequentially from the input. Operating with streams of nodes is unthinkable part of work holistic algorithms. Work efficiency of these algorithms is also dependent on the size of the input streams. In this work, we are about to reduce nodes in input streams of algorithm by using valid XML Schema to current XML document. This reduction works based on prefix and postfix optimization of twig pattern query. Twig pattern query is one of the most common types of queries, which handle holistic algorithms.

Keywords:

XPath, XML, XML tree, XML Schema, DTD, validation, oriented graph, pre-order tree traversal, twig pattern query, holistic algorithms, PathStack, TwigStack, GTPStack

Obsah

1 Úvod

XML(e**X**tensible **M**arkup **L**anguage) je rozšíriteľný značkovací jazyk, ktorý bol vyvinutý konzorciom W3C[1] ako pokračovanie jazyka SGML[2] a zovšeobecnenie jazyka HTML. Umožňuje jednoduché vytváranie konkrétnych značkovacích jazykov na rôzne účely a široké spektrum rôznych typov údajov.

Jazyk je určený predovšetkým na výmenu údajov medzi aplikáciami a na zverejňovanie dokumentov. Tento jazyk umožňuje opísať štruktúru dokumentu z hľadiska vecného obsahu jednotlivých častí a nezaoberá sa vzhľadom dokumentu alebo jeho časti. Vzhľad dokumentu sa potom definuje pripojeným štýlom. Ďalšou možnosťou je pomocou rôznych štýlov vykonať transformáciu do iného typu dokumentu alebo do inej štruktúry XML. Jazyk XML nemá žiadne preddefinované značky[3] a tiež jeho syntax je podstatne prísnejšia ale aj jednoduchšia ako syntax HTML.

1.1 XML

Sila XML je najmä v jeho hierarchickej štruktúre a pomerne jednoduchom spôsobe zápisu. Umožňuje opisovať alebo označovať ľubovoľné dáta a prenášať ich medzi rôznymi aplikáciami a platformami. Hlavnou ideou XML je oddelenie obsahu a vzhľadu dát. Spojovacím mostom je XSL[4], ktorý umožňuje vytvárať množstvo výstupných formátov. XML je formát súboru obsahujúci dáta. Tento formát je široko prijímaný ako štandard, ktorý môžeme použiť na akejkoľvek platforme a je používaný veľkým a stále rastúcim počtom aplikácií. V neposlednom rade, má XML širokú podporu v programovacích jazykoch.

Špecifikácia XML je popísaná odporučením konzorcia W3C. Tento jazyk nám dovoľuje vytvárať štruktúrované dokumenty veľmi flexibilným spôsobom. Dá sa použiť k vytváraniu dokumentov, ktoré sa zdanlivo podobajú dokumentom HTML, jazyk XML sa však od HTML líši a používa sa k zásadne iným účelom. Táto podobnosť má korene v rovnakých rodičoch – obidva jazyky sú odvodené od štandardu definície jazyka SGML. Ten sám o sebe nie je žiadny jazyk, ale iba spôsob definovania jazykov vyvinutých podľa jeho obecných princípov. Existuje tu však dôležitý rozdiel v tom, akým spôsobom sú jazyky XML a HTML odvodené od SGML. XML je podmnožina SGML – ľahšia verzia, ktorá bola oproti SGML zjednodušená, aby umožňovala použitie na sieti WWW, ale aj ako formát výmeny dát. Spomínané zjednodušenie robí zápis dokumentov vychádzajúcich z XML jednoduchším. Jazyk HTML je aplikáciou SGML – je to teda konkrétny jazyk, ktorý dodržuje štandard SGML.

[1] W3C – World Wide Web Consortium
[2] SGML – Standard Generalized Markup Language
[3] Tagy – názvy jednotlivých elementov
[4] XSL – eXtensible Stylesheet Language – jazyk popisujúci transformáciu XML do iného formátu

1.2 Pravidlá jazyka XML

Každý dokument XML sa skladá z kombinácie dát značiek a znakov. Značky dávajú dokumentu XML štruktúru, zatiaľ čo znaky predstavujú vlastný obsah. Všetky XML dokumenty spĺňajúce špecifikáciu XML musia dodržiavať isté pravidlá aby boli považované za správne štruktúrované.

- Každý element XML musí mať začiatočnú aj koncovú značku. „Prázdne“ elementy bez koncovej značky (<IMG> alebo <HR> v jazyku HTML) nie sú v XML povolené. Existuje však skrátený zápis, ktorý sa dá použiť v prípade, keď daný element neobsahuje žiadne dáta. Namiesto <empty></empty> môžeme použiť alternatívu <empty />. Tieto dve možnosti sa považujú za zhodné.
- Dokument XML musí obsahovať jediný pár značiek (skladajúci sa zo začiatočnej a koncovej značky), tzv. koreňový element dokumentu, v ktorom sú všetky ostatné elementy vložené. Na koreni nemôže byť viac elementov.
- Počiatočné a koncové značky každého elementu musia byť riadne vnorené, pričom vnorený element musí byť celý obsiahnutý vo svojom nadradenom elemente. Inými slovami, počiatočné a koncové značky vložených elementov sa nesmú prekrývať. To zaisťuje hierarchickú štruktúru dokumentu XML.
- Každý XML dokument by mal obsahovať hlavičku (výpis 1.1), tzv. XML deklaráciu. Tá sa typicky vyskytuje hneď na prvom riadku dokumentu. Táto deklarácia však nie je povinná. Je ohraničená ostrými zátvorkami s otáznikmi. Prvé je kľúčové slovo *„xml“*, za ktorým nasleduje verzia dokumentu, ktorá je povinná. Ďalej sa v tejto deklarácii môžu nachádzať atribúty kódovania dokumentu alebo atribút standalone, ktorý špecifikuje možnosť použitia externých súborov.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
```

Výpis 1.1 Deklarácia hlavičky XML súboru

Ak dokument XML spĺňa tieto pravidlá, potom je správne štruktúrovaný. Dokument XML sa označuje za platný, keď je správne štruktúrovaný a zároveň spĺňa požiadavky DTD[5], resp. XML Schema[6].

1.3 XML strom

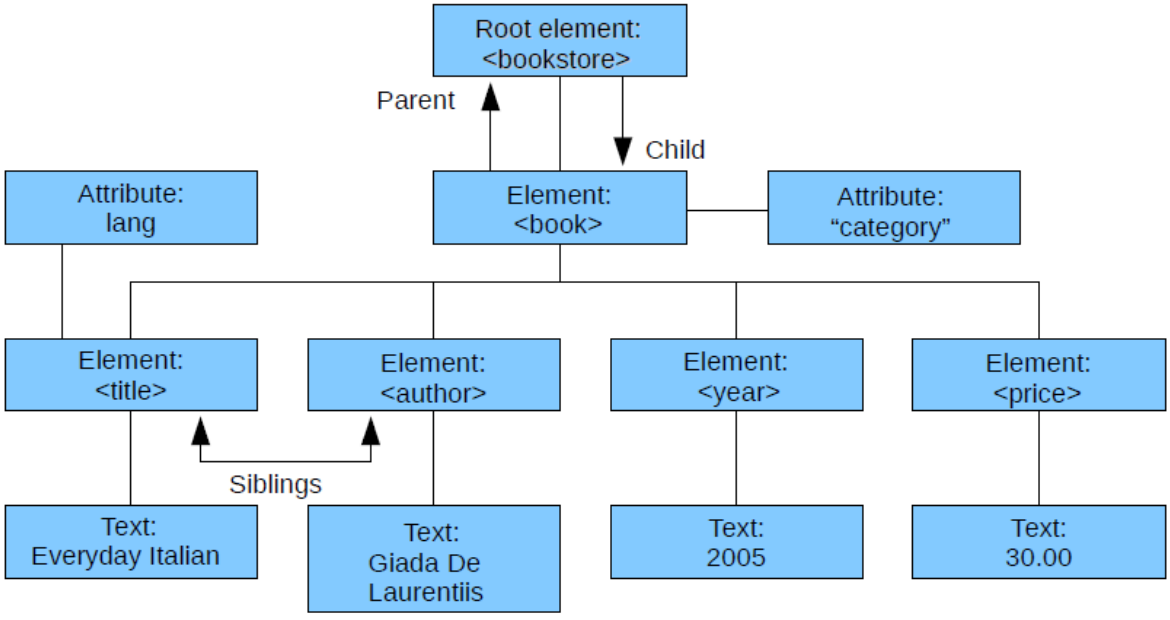
Na logickú štruktúru XML dokumentu sa môžeme pozerať ako na strom. Tento strom [1] začína koreňovým elementom, ktorý nesmie chýbať v žiadnom XML dokumente. Je to v podstate predok všetkých ďalších elementov, ktoré sa v dokumente nachádzajú. V XML strome sa nachádzajú všetky ostatné elementy, ďalej atribúty, prípadne ďalšie súčasti XML dát, ktoré sú spojené hranami. Na popis vzťahov medzi jednotlivými elementmi v strome sa pojmy ako rodič (parent), potomok (child), súrodenec (sibling).

[5] Document Type Definition – jazyk popisujúci štruktúru XML dokumentu
[6] XML Schema – popisuje štruktúru XML dokumentu, ktorý je tiež vo formáte XML

Na obrázku 1.1 máme možnosť vidieť XML strom, ktorý predstavuje XML dokument z výpisu 1.2. Pre lepšiu názornosť je v XML strome zahrnutý iba prvý XML uzol *book* a jeho potomkovia.

```
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada de Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="CHILDREN">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="WEB">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

Výpis 1.2 Ukážka XML dokumentu



Obrázok 1.1Ukážka XML Stromu

1.4 Validácia XML

Validácia XML je proces, počas ktorého sa kontrolujú dva aspekty dokumentu XML. Najprv sa overuje, či je dokument *well-formed*[7], čiže, či spĺňa základné syntaktické pravidlá tvorby,

[7] Well-formed – zaužívaný termín pre správne štruktúrovaný XML dokument

ktoré sú rovnaké pre všetky XML dokumenty. Ďalej sa kontroluje, či dokument spĺňa pravidlá, ktoré sú popísané príslušným DTD dokumentom alebo XML Schema.

1.4.1 Document Type Definition (DTD)

Prvý jazyk, ktorý popisuje štruktúru XML dokumentu je DTD [2]. Ten definuje elementy, ktoré môžu byť v dokumente zahrnuté, ďalej atribúty, ktoré môžu elementy obsahovať, a tiež poradie a vnorenie jednotlivých elementov. Tento jazyk má svoju vlastnú unikátnu syntax odvodenú od jazyka SGML. DTD sa deklaruje pomocou kľúčového slova DOCTYPE hneď pod hlavičkou deklarácie dokumentu XML, pričom rozlišujeme dva typy deklarácie:

- *Inline deklarácia* - v tomto prípade sa definuje DTD priamo v dokumente XML a to v hranatých zátvorkách.

```
<?xml version="1.0"?>
<!DOCTYPE documentelement [definicia_DTD]>
```

Výpis 1.3 Inline deklarácia DTD

- *Externá deklarácia* - znamená, že deklarácia DTD je umiestnená v samostatnom dokumente, ku ktorému musíme uviesť systémovú cestu.

```
<?xml version="1.0"?>
<!DOCTYPE documentelement SYSTEM "definiciaDTD.dtd">
```

Výpis 1.4 Externá deklarácia DTD

Vo výpise 1.5 je zobrazená inline deklarácia DTD dokumentu spolu s XML dátami o knižnici. Koreňový element *bookstore* obsahuje obsahuje element *name* a 1 alebo viac výskytov elementu *topic*. Každý element *topic* obsahuje element *name* a 0 alebo viac elementov *book*. Element *book* obsahuje elementy *title* a *author* a atribút *isbn.* Element *name* má priradený typ PCDATA (Parsed character data). *Title* a *author* majú typ CDATA (Character data).

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE bookstore [
  <!ELEMENT bookstore (name,topic+)>
  <!ELEMENT topic (name,book*)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT book (title,author)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT isbn (#PCDATA)>
  <!ATTLIST book isbn CDATA "0">
]>
<bookstore>
  <name>Book Store Name</name>
  <topic>
    <name>XML</name>
    <book isbn="123-456-789">
      <title>DTD Example</title>
      <author>Marek Lakostik</author>
    </book>
  </topic>
</bookstore>
```

Výpis 1.5 Ukážka inline DTD deklarácie spolu s dátami

DTD má jednoduchú syntax a nevyžaduje žiadne náročné písanie kódu. Avšak podporuje málo dátových typov a nepodporuje menné priestory. DTD tiež nie je príliš komplexné - všetko čo sa dá napísať v DTD, dá sa napísať aj v pokročilejšej technológii XML Schema (kapitola 1.4.2). V opačnom garde to ale neplatí. Keďže je ale XML Schema stále relatívne nová technológia a stále sa rozvíja, nie všetky parsery ju podporujú. Naopak, DTD sa za čas svojej existencie stal široko podporovanou technológiou.

1.4.2 XML Schema

XML Schema [3] poskytuje omnoho silnejšie prostriedky na tvorbu definícií a limitácií dokumentu XML. Sám dokument XML Schema je napísaný v jazyku XML. Jej základný cieľ je poskytnúť objektovo orientovaný prístup ku definovaniu formátu XML dokumentu. Poskytuje užívateľom omnoho väčšiu skupinu dátových typov oproti DTD, ktoré zahrňujú základné programátorské typy ako sú *integer*, *byte*, *string*, *float* alebo *date*. Ďalšou veľkou výhodou je, že kombinovaním týchto typov spolu s ďalšími operátormi a modifikátormi môžeme vytvoriť svoj vlastný užívateľský typ, tzv. *komplexný typ*. Tento užívateľský typ potom môžeme priradiť ktorémukoľvek elementu ako jeho dátový typ.

1.4.2.1 Koreňový XML Schema element

Keďže XML Schema [4] je v podstate XML, tak aj táto technológia má svoj koreňový element. Ten nesie názov *schema* a musí ho obsahovať každá deklarácia schémy. Tento element obsahuje poväčšine niekoľko voliteľných atribútov. Vo výpise 1.6 je znázornené, ako *schema* element môže vyzerať.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.w3schools.com"
           xmlns="http://www.w3schools.com"
           elementFormDefault="qualified">
  ...
  ...
</xs:schema>
```

Výpis 1.6 Deklarácia koreňového elementu XML Schema

- Tento fragment:

  ```
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  ```

  hovorí, že elementy a dátové typy použité v schéme pochádzajú z menného priestoru *"http://www.w3.org/2001/XMLSchema"*. Ďalej sa tu uvádza, že elementy a dátové typy, ktoré pochádzajú z tohto menného priestoru by mali obsahovať prefix *xs:*.

- Tento fragment:

  ```
  targetNamespace="http://www.w3schools.com"
  ```

  signalizuje, že elementy definované v tejto schéme pochádzajú z menného priestoru *"http://www.w3schools.com"*.

- Tento fragment:

```
xmlns="http://www.w3schools.com"
```

  popisuje implicitný menný priestor.

- Tento fragment:

```
elementFormDefault="qualified"
```

  určuje, že každý element použitý v XML dokumente, ktorý bol deklarovaný v danej schéme musí mať kvalifikovaný menný priestor.

1.4.2.2 XML Schema element

V XML Schema delíme elementy na tzv. *Simple* a *Complex* elementy.

- *Simple* element, v preklade jednoduchý element, je element, ktorý obsahuje iba text. Nemôže už teda obsahovať ďalšie elementy alebo atribúty. Avšak toto obmedzenie môže byť trochu mätúce. Text môže byť totiž rôznych typov, ktoré sú zahrnuté v definícii XML Schema (*boolean*, *string*, *date*, atd.), ale môže to byť napríklad aj užívateľsky definovaný typ.
- *Complex* (komplexný) element predstavuje element, ktorý obsahuje ďalšie elementy alebo atribúty. Podľa XML Schema existujú štyri druhy komplexných elementov:
  - Prázdne elementy
  - Elementy, ktoré obsahujú iba ďalšie elementy
  - Elementy, ktoré obsahujú iba text
  - Elementy, ktoré obsahujú ďalšie elementy aj text

Vo výpise 1.7 sme vytvorili komplexný typ *personinfo*. Ten obsahuje elementy *firstname* a *lastname*, ktoré sú ohraničené elementom *xs:sequence.* Element *sequence* určuje, že elementy v ňom uzavreté sa musia v XML dokumente objaviť presne v tom poradí, v akom sú v ňom deklarované. Deklarovaný komplexný typ môže využiť viac elementov, ako v tomto prípade, keď elementy *employee*, *student* a *member* sú komplexného typu *personinfo*.

```
<xs:element name="employee" type="personinfo"/>
<xs:element name="student" type="personinfo"/>
<xs:element name="member" type="personinfo"/>

<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Výpis 1.7 Využitie komplexného typu v XML Schema

1.4.2.3 Vlastnosti XML Schema elementov

XML Schema elementy (*simple* aj *complex*) disponujú rôznymi obmedzeniami, ktorými môžeme lepšie zachytiť charakter dát.

- *Default* a *Fixed* atribúty

```
<xs:element name="color" type="xs:string" default="red"/>
```

Výpis 1.8 Definícia elementu color s predvolenou hodnotou red

Default hodnota je priradená elementu automaticky v prípade, že element nemá špecifikovanú žiadnu hodnotu. Naopak *Fixed* hodnota je priradená elementu automaticky a nie je mu možné priradiť inú hodnotu.

- Obmedzenie na hodnoty

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="120"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Výpis 1.9 Definícia elementu age s minimálnou a maximálnou hodnotou

Pomocou XML Schema elementov *minInclusive* a *maxInclusive* môžeme obmedzovať hodnoty elementov. Tieto elementy sú súčasťou elementu *restriction*.

- Obmedzenie na sériu znakov

```
<xs:element name="initials">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[A-Z][A-Z][A-Z]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Výpis 1.10 Element initials akceptuje ako hodnotu iba 3 veľké písmená

Na limitovanie obsahu elementov na určený počet čísiel alebo písmen sa používa element s názvom *pattern*, ktorý je tak isto súčasťou elementu *restriction*.

- Obmedzenie na dĺžku

```
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:length value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Výpis 1.11 Hodnota elementu password musí mať dĺžku 8

Element *restriction* umožňuje limitovanie dĺžky hodnoty elementu. Na tento účel sa používajú jeho vnorené elementy *length*, *maxLength*, *minLength*.

- *Occurence* indikátory

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="full_name" type="xs:string"/>
      <xs:element name="child_name" type="xs:string" maxOccurs="10"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Výpis 1.12 Element person môže mať maximálne 10 elementov child_name

Výskytové indikátory určujú koľko krát sa môže v XML dokumente daný element objaviť. Vlastnosť *maxOccurs* určuje maximálny počet a *minOccurs* určuje minimálny počet. Kľúčové slovo *unbounded*, určuje že počet výskytov je neohraničený. V prípade, že nie sú tieto dve vlastnosti zadané, XML Schema implicitne predpokladá, že obe tieto vlastnosti nadobúdajú hodnotu 1.

- *Choice* element

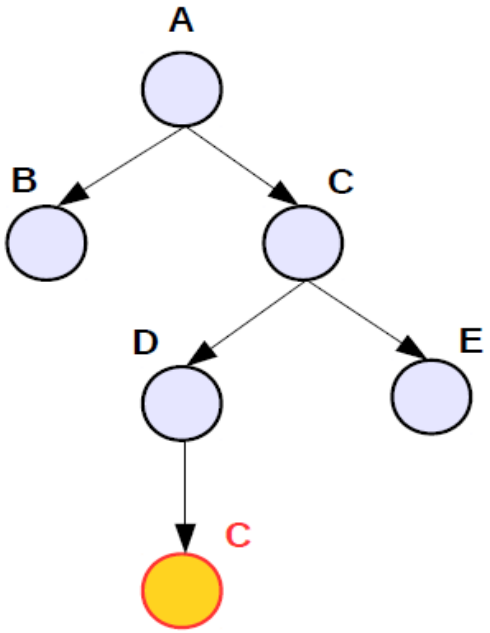
```
<xs:element name="person">
  <xs:complexType>
    <xs:choice>
      <xs:element name="employee" type="xs:string"/>
      <xs:element name="member" type="xs:string"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Výpis 1.13 Deklarácia XML Schema elementu choice

XML Schema *choice* element umožňuje aby iba jeden (alebo niektoré - závisí od formy deklarácie) z elementov, ktoré uzatvára, bol prítomný v príslušnom XML dokumente. Napríklad, podľa výpisu 1.13 vyššie, v XML dokumente smie mať element person buď potomka *employee* alebo potomka *member*, nie však oboch zároveň.

1.4.2.4 Rekurzia v XML Schema

Keďže XML dokumenty môžu obsahovať rekurzívne vnorené uzly, aj XML Schema umožňuje definovať rekurzívne elementy. Na obrázku 1.2 je zobrazený orientovaný graf, ktorý predstavuje schému dokumentu, pričom rekurzívny element je vyznačený žltou farbou. V XML dokumente, validnom podľa tejto schémy, každý XML uzol D ktorý je rodičom uzlu C (odpovedá v schéme rekurzívnemu elementu C), môže, ale nemusí znova obsahovať nové rekurzívne vnorenie XML uzlov C,D,E (výpis 1.14.).



Obrázok 1.2 Ukážka XML Schema orientovaného grafu

```
<?xml version="1.0" encoding="utf-8"?>
<A xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="recSchema.xsd">
  <B></B>
  <C>
    <D>
      <C>
        <D>
          <C>
            <D></D>
            <E></E>
          </C>
        </D>
        <E></E>
      </C>
    </D>
    <E></E>
  </C>
</A>
```

Výpis 1.14 Ukážka definície XML dokumentu podľa schémy z obrázku 1.2

Deklarácia rekurzívnych elementov sa dá v XML Schema dosiahnuť dvoma spôsobmi:

- Definícia rekurzie pomocou odkazu (výpis 1.15), t.j. pomocou atribútu *ref*, ktorý patrí elementu. Ako *ref* atribút sa udáva meno elementu, na ktorý má aktuálny element odkazovať.
- Definovať rekurziu môžeme v prostredí XML Schema aj pomocou vlastného užívateľského typu (výpis 1.16). Aby bol element s vlastným užívateľským typom rekurzívny, musí v sebe obsahovať element, ktorý je znova tohto typu.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="A">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="B"/>
        <xs:element ref="C"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="C">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="D">
          <xs:complexType>
            <xs:sequence>
              <xs:element ref="C" minOccurs="0"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="E"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Výpis 1.15 Definícia rekurzívneho dokumentu z príkladu 1.14 pomocou odkazu

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="A">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="B"/>
        <xs:element name="C" type="recursiveType"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="recursiveType">
    <xs:sequence>
      <xs:element name="D">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="C" type="recursiveType" minOccurs="0"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="E"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Výpis 1.16 Definícia rekurzívneho dokumentu z príkladu 1.14 pomocou vlastného užívateľského typu

Je dôležité, aby sa v oboch prípadoch, pri definícii rekurzívneho elementu deklaroval aj jeho atribút *minOccurs* s priradenou hodnotou 0, pretože XML Schema implicitne predpokladá práve jeden výskyt každého elementu v prípade, že ho nemá definovaný. Bez tohto atribútu by sa nedala rekurzia v schéme nikdy ukončiť.

1.4.2.5 Validný dokument ku XML Schema

Tak ako sme v kapitole 1.4.1 vytvorili príklad schémy knižnice v DTD, pre porovnanie sme vytvorili rovnakú v XML Schema, ktorú máme možnosť vidieť vo výpise 1.17. Môžeme vidieť, že na rozdiel od DTD je možné deklarovať konkrétny typ daného elementu. Pomocou elementu obmedzenia (*restriction*) sme mohli presne definovať tvar atribútu *isbn*, ktorý musí spĺňať podmienku, že sú to trojice čísiel oddelené pomlčkami. U elementov *topic* a *book* sme určili dolnú a hornú hranicu počtu ich výskytov. Výpis 1.18 ilustruje XML dokument validný ku XML Schema, ktorá je znázornená vo výpise 1.17. Prepojenie XML dokumentu s XML Schema zabezpečuje atribút *schemaLocation*, ktorý ako prvý parameter akceptuje menný priestor a druhý parameter je názov dokumentu, kde sa schéma nachádza. V tomto prípade je teda deklarácia XML Schema oddelená od XML dokumentu.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.w3schools.com"
           xmlns="http://www.w3schools.com"
           elementFormDefault="qualified">
  <xs:element name="bookstore">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="topic" minOccurs="1" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" type="xs:string"/>
              <xs:element name="book" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="title" type="xs:string"/>
                    <xs:element name="author" type="xs:string"/>
                  </xs:sequence>
                  <xs:attribute name="ISBN-type">
                    <xs:simpleType>
                      <xs:restriction base="xs:string">
                        <xs:pattern value="\d{3}-\d{3}-\d{3}"/>
                      </xs:restriction>
                    </xs:simpleType>
                  </xs:attribute>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Výpis 1.17 XML Schema knižnice

```
<?xml version="1.0"?>
<bookstore xmlns="http://www.w3schools.com"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.w3schools.com
bookstore_schema.xsd">
  <name>Book Store Name</name>
  <topic>
    <name>XML</name>
    <book ISBN-type="123-456-789">
      <title>XSD Example</title>
      <author>Marek Lakostik</author>
    </book>
  </topic>
</bookstore>
```

Výpis 1.18 Dokument validný ku XML Schema z výpisu 1.17

2 Dotazovanie nad XML

2.1 XML Path Language (XPath)

Bežnou úlohou pre mnoho aplikácií založených na XML je identifikovať určité časti XML dokumentu. Na miesto toho, aby si musela každá aplikácia vytvárať a používať svoju vlastnú metódu dotazovania XML dokumentu, W3C vyvinulo Xml Path Language (XPath). Základnú myšlienku XPath vyjadruje už samotné meno tohto jazyka. Je ňou adresovanie časti XML dokumentu pomocou výrazov ciest. Tieto výrazy vyzerajú podobne ako výrazy pri práci s tradičným počítačovým súborovým systémom. XPath zahrňuje vyše 100 vstavaných funkcií. Sú to napríklad funkcie, ktoré spracúvajú reťazce, číselné hodnoty, logické hodnoty, dátum a čas, a ďalšie.

XPath [5], ktorý je súčasťou dotazovacieho jazyka XQuery, umožňuje definovať výrazy ciest pre výber množiny uzlov z dokumentu. Dokáže ale aj filtrovať uzly stromu pomocou relácií medzi nimi pre priechod XML stromom. Dotaz sa skladá z výrazov, ktoré majú tvar *osa::značka[podmienka]*. Osa určuje typ relácie medzi jednotlivými uzlami. V tabuľke 2.1 sú popísané existujúce typy relácií medzi uzlami.

Osa	**Preklad**	**Výsledok**
ancestor	predok	Všetky uzly ležiace na ceste od koreňa k uzlu *u*
ancestor-or-self	predok alebo ja	Všetky uzly ležiace na ceste od koreňa k uzlu *u* (vrátane)
attribute	Atribút	Všetky atribúty uzlu *u*
child	dieťa	Všetky uzly, ktoré sú deti uzlu *u*
descendant	potomok	Všetky uzly, pre ktoré je *u* predok
descendant-or-self	potomok alebo ja	Všetky uzly, pre ktoré je *u* (vrátane) predok
following	nasledujúci	Všetky uzly, nasledujúce uzol *u*, okrem potomkov
following-sibling	nasledujúci súrodenci	Všetky uzly, nasledujúce uzol *u* (vrátane), okrem potomkov
namespace	menný priestor	Všetky menné priestory uzlu *u*
parent	rodič	Prvý uzol na ceste od uzlu *u* smerom ku koreňu
preceding	predchádzajúci	Všetky uzly, predchádzajúce uzol *u* okrem predkov
preceding-sibling	Predchádzajúci súrodenci	Všetky uzly, predchádzajúce uzol *u* (vrátane), okrem predkov
self	ja	Aktuálny uzol *u*

Tabuľka 2.1 Výpis osí, ktoré podporuje jazyk XPath

Značka popisuje meno uzlu a podmienka je určité obmedzenie uzlu. Jednotlivé výrazy sú oddelené / alebo //. Znak / predstavuje skrátený zápis osy *child::* a znak // skrátený zápis osy *descendant::*. Tabuľka 2.2 zobrazuje najpoužívanejšie výrazy.

Výraz	**Popis**
meno_uzlu	Vyberie všetky uzly s názvom *meno_uzlu*
/	Vyberá od koreňového uzlu, tiež skrátený zápis osy *child::*
//	Vyberá uzly od aktuálneho uzla, ktoré spĺňajú kritérium, pričom nezáleží, kde sa nachádzajú, tiež skrátený zápis osy *descendant::*
.	Vyberá aktuálny uzol
..	Vyberá rodiča aktuálneho uzlu
@	Vyberá atribúty

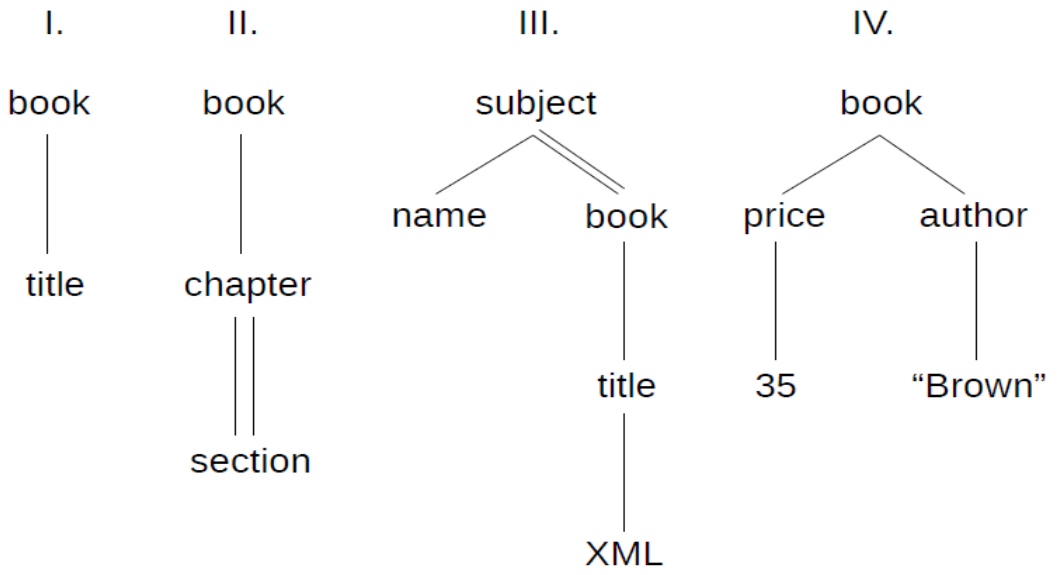
Tabuľka 2.2 Výrazy v jazyku XPath

Normálne je XML dokument modelovaný ako usporiadaný strom. Vnútorné uzly reprezentujú elementy a atribúty v XML dokumente. Listové uzly reprezentujú dáta, čo je text uložený v elemente alebo v atribúte. Hrany v strome teda vyjadrujú vzťahy element-subelement, element-atribút, element-hodnota, a atribút-hodnota. Dva uzly v strome spojené hranou sú vo vzťahu rodič-dieťa (parent-child označované skratkou PC), a dva uzly nachádzajúce sa na rovnakej ceste sú vo vzťahu predok-potomok (ancestor-descendant – označované ako AD).

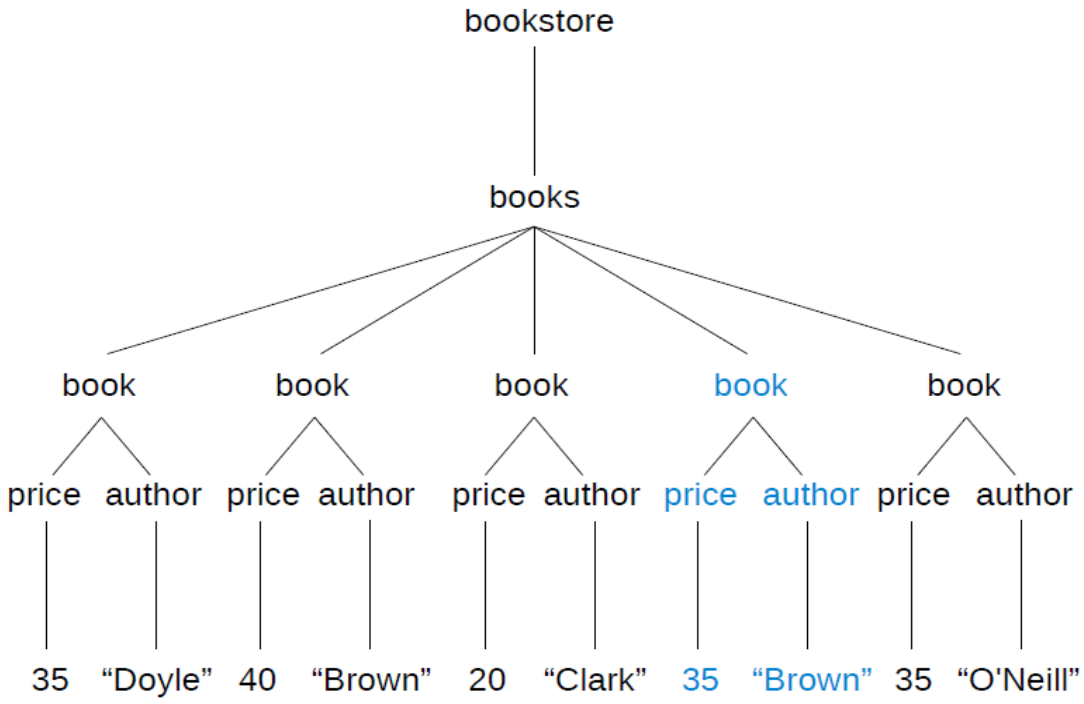
Dotaz vo väčšine štandardných XML dotazovacích jazykoch sa dá tiež modelovať ako strom, pričom sa o ňom v literatúre často zmieňuje ako o *twig pattern*, v preklade vetvený dotaz. Obzvlášť XPath dotaz je normálne modelovaný ako vetvený dotaz. Proces, počas ktorého sa vyhľadávajú výskyty vetveného dotazu v XML dokumente sa označuje pojmom *twig pattern matching*.
Formálne je dotazovanie nad XML dokumentom definované takto:

Máme daný XML dokument *D* a vetvený XML dotaz *Q*. Cieľom je nájsť všetky výskyty *Q* v *D* také, že *(i)* predikáty uzlov dotazu sú reprezentované odpovedajúcimi elementmi XML dokumentu, a *(ii)* PC a AD vzťahy medzi uzlami dotazu sú splnené medzi príslušnými elementmi dokumentu. Výsledok dotazu *Q* s *n* uzlami môže byť reprezentovaný zoznamom n-tic, kde sa každá n-tica $(q_1, q_2, \ldots q_n)$ skladá z elementov dokumentu presne vyhovujúcemu dotazu *Q* v *D*.

Na obrázku 2.1 sú znázornené príklady XPath dotazov a k nim príslušné vetvené dotazy. Obrázok 2.2 ilustruje príklad stromovej prezentácie XML dokumentu, v ktorom je znázornený výsledok dotazu IV z obrázku 2.1 .



Obrázok 2.1 Typy dotazov: I. book/title; II. book/chapter//section; III. subject[//book/title=“XML“]/name; IV. book[/price=35]/author=“Brown“



Obrázok 2.2 XML strom a vyznačenie odpovede na dotaz IV z obrázku 2.1

2.2 Značkovacie schémy

Algoritmy, ktoré spracúvajú dotazovanie nad dokumentmi XML pomocou jazyka XPath, sú založené na stromovej štruktúre. K efektívnemu využitiu tejto štruktúry môžeme zaviesť kódovanie prvkov v rámci stromu, nech už sa jedná o elementy, atribúty alebo textové uzly. Dobre očíslovaný XML dokument umožňuje efektívnejšie dotazovanie, pretože číslovanie

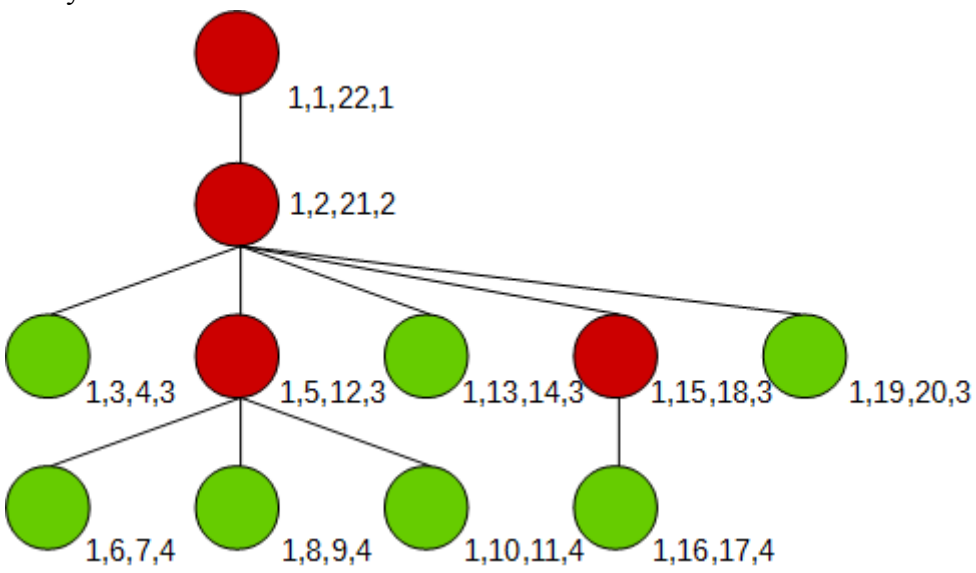
nám zaistí unikátnu identifikáciu pre všetky prvky XML dokumentu vrátane rozlišovania vzťahov. Môžeme teda jednoducho rozlíšiť vzťah medzi dvoma vybranými uzlami.

Väčšina algoritmov, ktoré vykonávajú XPath dotazy, používa nejakú predspracovanú dátovú štruktúru, kde je uložené schéma dokumentu. Zväčša to býva nejaký druh invertovaného zoznamu, kde sa efektivita zvyšuje a klesá podľa druhu jeho implementácie. Kľúčom invertovaného zoznamu je uzol XML stromu dokumentu. Spolu s týmto kľúčom sa ukladajú aj čísla uzlu odpovedajúce použitému značkovaciemu schématu.

Existuje množstvo rozličných značkovacích schém [6] s rôznymi vlastnosťami. Spoločné majú však to, že nám umožňujú zistiť základné vzťahy medzi uzlami (rodič, predok, atď.). Rozlišujeme dva hlavné typy značkovacích schém:

- **Značkovacie schéma založené na elementoch (Element labeling scheme)**

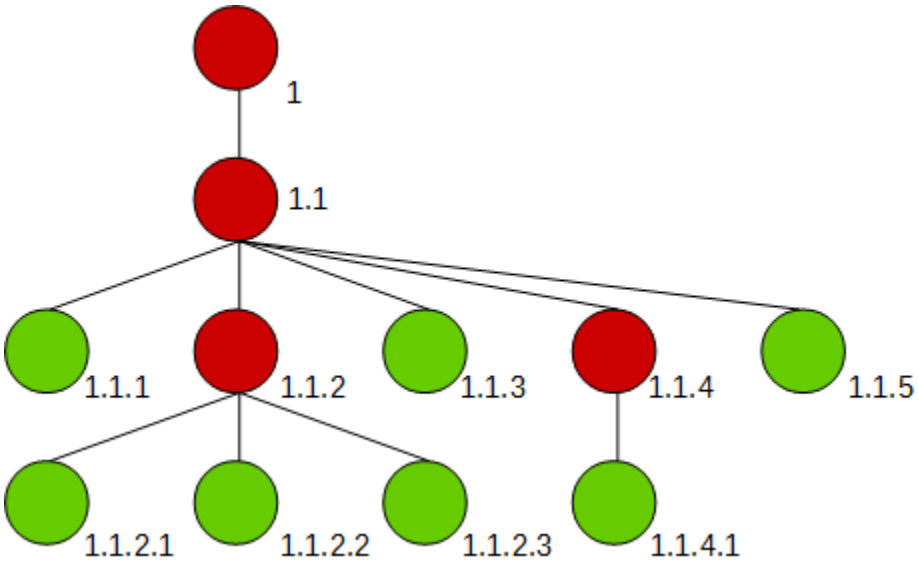
  Značky sú reprezentované kódovaním, ktoré má pre všetky uzly dokumentu rovnakú dĺžku (obrázok 2.3). Označenie každého prvku sa skladá zo štvorice hodnôt (*docID, leftPos, rightPos, level)*, kde *docID* jednoznačne identifikuje XML dokument, v ktorom sa značka nachádza. Uzly z rôznych dokumentov sa budú v tejto hodnote líšiť a naopak, v prípade rovnosti tejto hodnoty sa budú nachádzať v rovnakom dokumente. Skutočnosť, že pri prechode XML stromom prejdeme každým uzlom dva krát, sa využíva pre priradenie tzv. intervalového ohodnotenia hodnotám *leftPos* a *rightPos*. Označenie *leftPos* je pozícia slova v rámci XML dokumentu počítaná od začiatku daného dokumentu. Táto hodnota sa vytvára pri prvom prechode cez daný uzol. Naopak hodnota *rightPos* je vytvorená pri druhom prechode daným uzlom. Na implementáciu je teda vhodné použiť počítadlo. Posledná hodnota *level* nám ukladá hĺbku, v ktorej sa uzol nachádza, pričom koreňový uzol má zvyčajne hĺbku 1. Nevýhodou je, že pri hoci len malej zmene XML dokumentu sa musí preznačkovať celý XML dokument.



Obrázok 2.3 Schéma XML dokumentu kódované pomocou Element labeling scheme - predpokladám, že docID je 1

- **Značkovacie schéma založené na cestách (Path labeling scheme)**

  V tomto prípade (obrázok 2.4) majú značky premennú dĺžku kódovania. Značkovacie schéma tohto typu je založené na priechode XML dokumentu do šírky. Najprv je označený rodič a následne všetky jeho deti tak, že obsahujú označenie svojho rodiča (vrátane všetkých jeho predkov) a k tomu obsahuje aj vlastnú pridanú unikátnu značku. Jednotlivé označenia predkov a aktuálneho uzlu bývajú oddelené znakom „.“. Výhodou je, že pri zmene XML dokumentu sa nemusí preznačkovať celý dokument ale iba daná časť. Naopak, nevýhodou je komplikované ukladanie značiek kvôli variabilite dĺžok značiek.



Obrázok 2.4 Schéma XML dokumentu kódované pomocou Path labeling scheme

3 XPath algoritmy vykonávajúce operáciu Twig Join

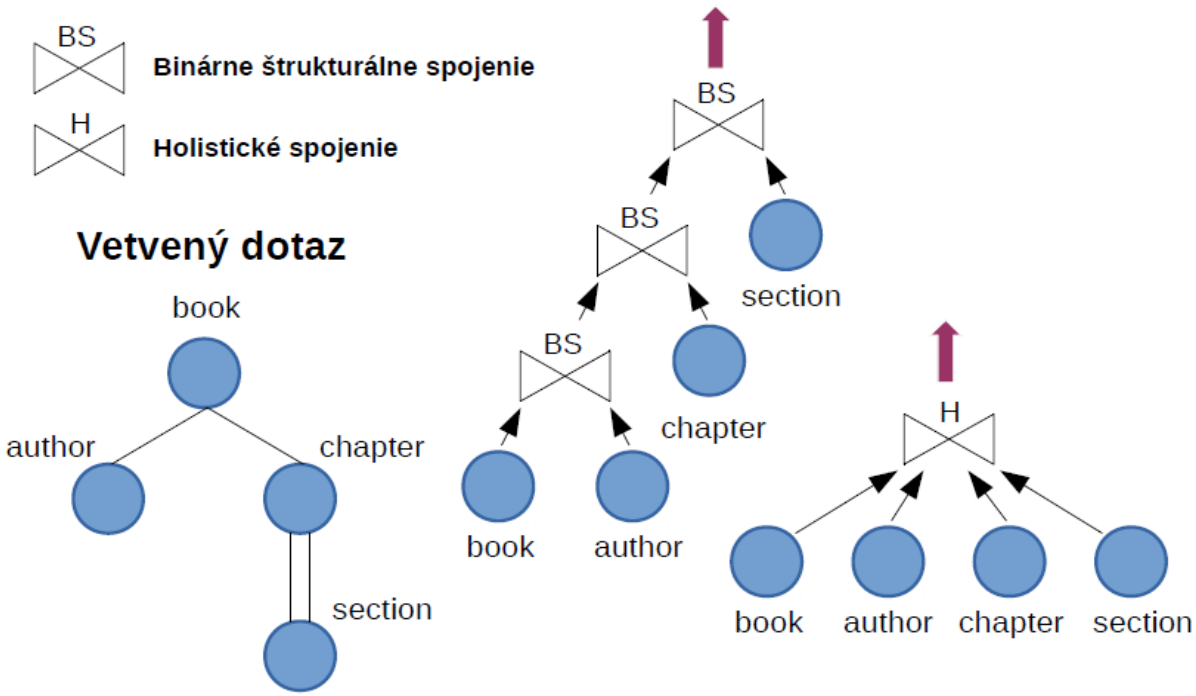
Na vykonávanie XPath dotazov sa okrem iných používajú aj tzv. Twig Join algoritmy. Existuje ich celá rada, v zásade sa ale líšia v operácii spojenia (obrázok 3.1). Podľa tejto vlastnosti ich delíme na:

- **Algoritmy s binárnym štrukturálnym spojením**

  Tieto algoritmy rozdeľujú dotaz XPath na dvojice výrazov, v čom spočíva ich hlavná nevýhoda. Tou je, že môžu produkovať veľké medzivýsledky, ktoré nebudú súčasťou konečného výsledku. S tým je spojená nie veľká efektivita týchto algoritmov. Zástupcami týchto algoritmov sú napríklad algoritmus MPMGJN[8] , ktorý bol jedným z prvých algoritmov navrhnutých na dotazovanie sa nad XML dokumentom pomocou XPath. Ďalšími zástupcami tejto skupiny algoritmov sú napríklad algoritmy patriace do rodiny Stack-tree.

- **Algoritmy s holistickým spojením**

  Táto skupina algoritmov sa líši od prechádzajúcej tým, že berie v úvahu všetky výrazy XPath dotazu zároveň, čím môžu dosahovať lepších výsledkov z pohľadu optimality. Celková náročnosť na pamäť, spojená s ukladaním medzivýsledkov môže byť menšia, rovnako ako aj celková časová náročnosť. Ďalej v tejto práci pracujeme s touto skupinou algoritmov.



Obrázok 3.1 Porovnanie vykonania dotazu algoritmom s binárnym štrukturálnym resp. holistickým spojením

[8] Multi-Predicate-Merge-Join

3.1 Funkcie a dátové štruktúry

Na úvod si ozrejmime niektoré základné operácie, ktoré využívajú holistické algoritmy pri práci s uzlami vetveného dotazu počas vykonávania dotazu. V príkladoch q predstavuje uzol dotazu.

isLeaf(q): Bool – funkcia vráti logickú hodnotu, na základe toho, či je aktuálny uzol dotazu listom.

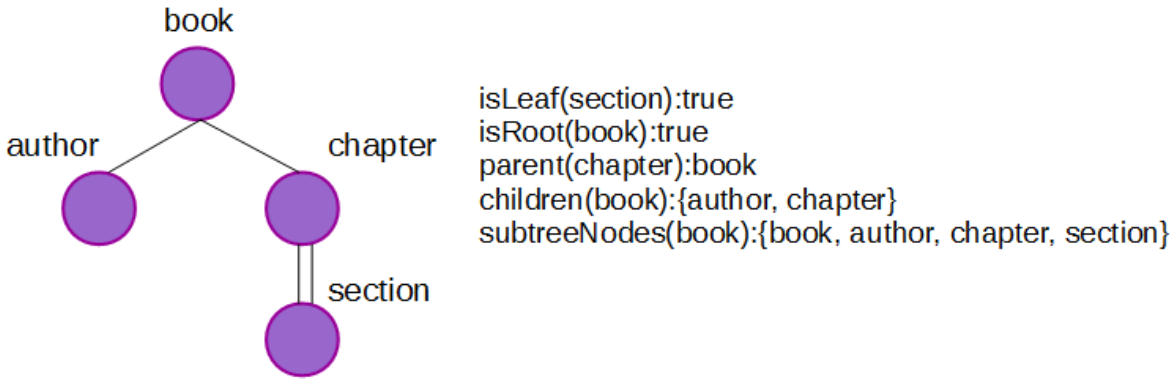
isRoot(q): Bool – funkcia vráti logickú hodnotu, na základe toho, či je aktuálny uzol dotazu koreňom.

parent(q): q' – funkcia vráti uzol dotazu *q'*, ktorý je rodičom aktuálneho uzlu dotazu q.

children(q): $\{q_1,..., q_n\}$ – funkcia vráti všetky uzly dotazu, ktoré sú deťmi aktuálneho uzlu dotazu.

subtreeNodes(q): $\{q,q_1,..., q_n\}$ – funkcia vracia uzol dotazu q a všetkých jeho potomkov.

Na obrázku 3.2 máme znázornený vetvený dotaz a k nemu príklady pre všetky spomenuté funkcie.



Obrázok 3.2 Znázornenie dotazu a k nemu príklady operácii, ktoré používajú holistické algoritmy

3.1.1 Streamy

K jednotlivým uzlom dotazu XPath je priradený tzv. *stream T*, v preklade prúd. Stream *T* obsahuje množinu spracovaných uzlov XML dokumentu. Táto množina uzlov je vytvorená vo forme zvolenej značkovacej schémy. V ďalšom texte budeme predpokladať, že použitá značkovacia schéma je Element Labeling Scheme (kapitola 2.2). Uzly v streamoch sú zoradené prvotne podľa *docID* a druhotne podľa *leftPos*.
Aj streamy majú operácie s ktorými pracujú:

end(T_Q)*:* vráti logickú hodnotu, podľa toho, či je ukazateľ streamu T_Q na konci.

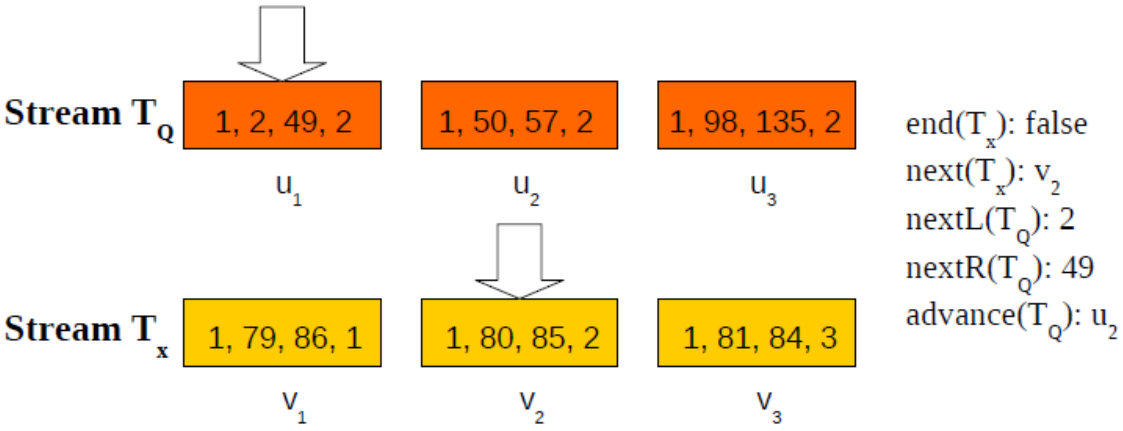
advance(T_Q)*:* posúva ukazateľ streamu T_Q o jeden uzol doprava.

next(T_Q): vráti aktuálnu značku ukazateľa streamu T_Q.

nextL(T_Q): vracia *leftPos* aktuálnej značky ukazateľa streamu T_Q.

nextR(T_Q): vracia *rightPos* aktuálnej značky ukazateľa streamu T_Q.

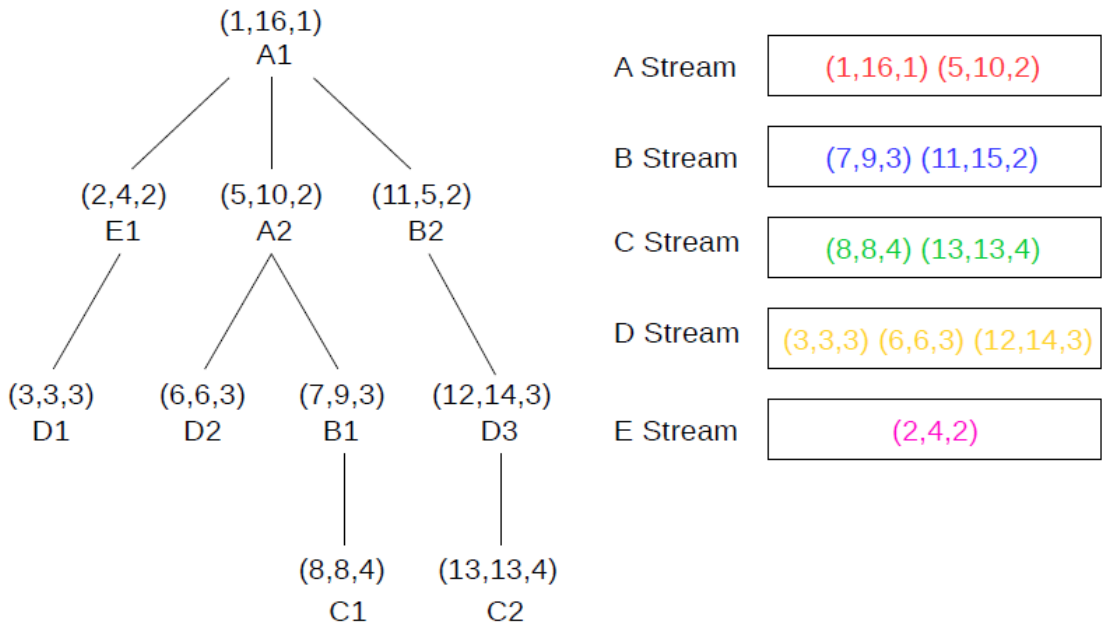
Na obrázku 3.3 máme možnosť vidieť, ako fungujú prúdové operácie.



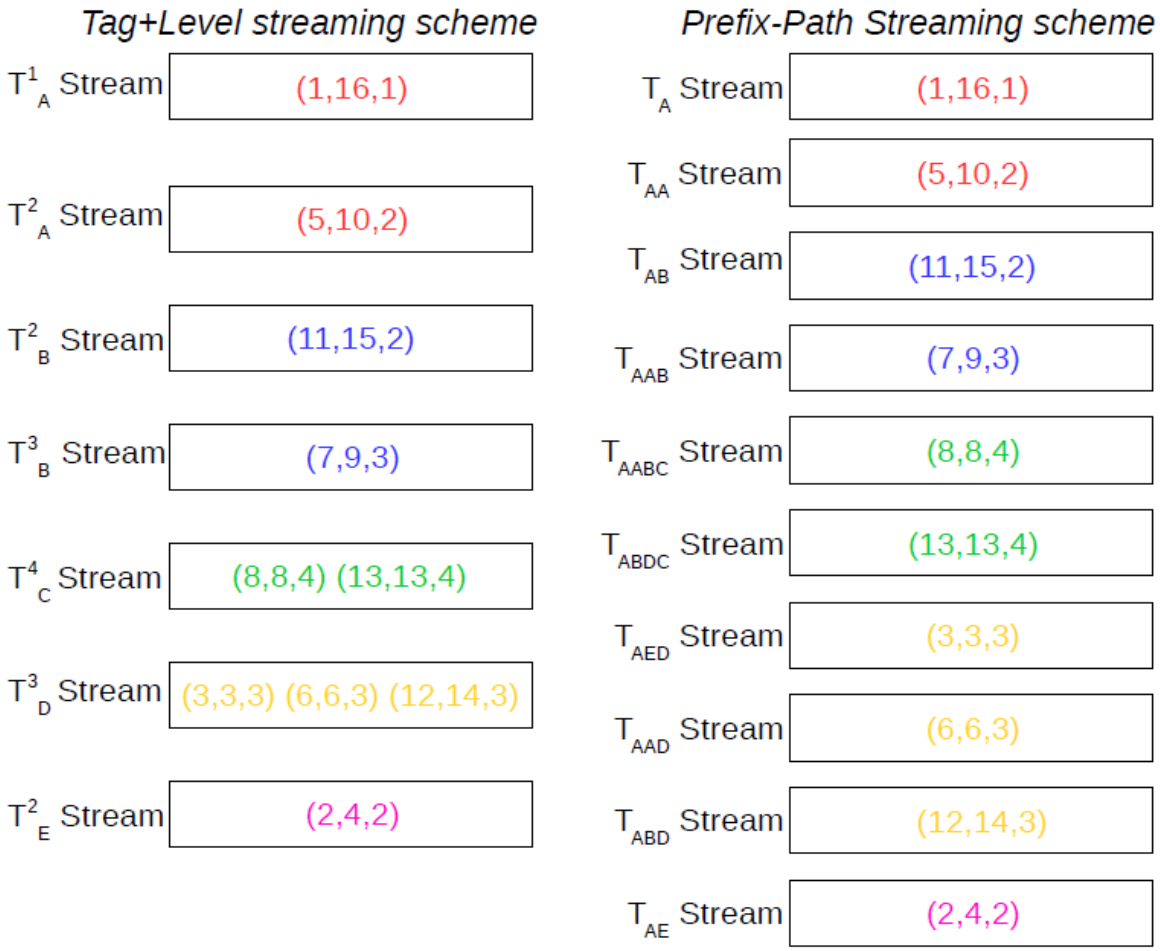
Obrázok 3.3 Ukážka operácií s prúdmi a ich výsledky

Holistické algoritmy v priebehu vykonávania dotazu čítajú tieto streamy jednotlivých uzlov vetveného dotazu, pokiaľ ich neprečítajú celé. Forma, ktorou sa vytvárajú jednotlivé streamy pre každý uzol vetveného dotazu predstavuje tzv. streaming scheme [7]. Často používané metódy sú:

- *Tag streaming scheme* – Táto metóda (obrázok 3.4) zoskupuje do rovnakých streamov všetky uzly XML dokumentu s rovnakým názvom. Pre príklad, stream T_A bude zoskupovať všetky elementy s názvom A.
- *Tag+level streaming scheme* – Táto metóda (obrázok 3.5) je odvodená od predchádzajúcej, ale navyše obsahuje ešte informáciu, ktorou je hĺbka (level) daného uzlu XML dokumentu. To znamená, že do jednoho streamu budú priradené všetky elementy, ktoré majú rovnaký názov elementu a zároveň majú rovnakú hĺbku v stromovej prezentácii dokumentu XML.
- *Prefix-Path streaming scheme (PPS)* – Táto metóda (obrázok 3.5) je založená na prefixovej ceste uzlov v stromovej štruktúre XML dokumentu. Prefixová cesta je cesta od koreňa dokumentu k danému elementu. Do jedného streamu budú zaradené všetky uzly XML dokumentu s rovnakou prefixovou cestou.



Obrázok 3.4 XML dokument a jeho rozdelenie do streamov podľa Tag streaming scheme



Obrázok 3.5 Rozdelenie značiek pre XML dokument z obrázku 3.4 do Tag+level a PPS streaming schém

3.1.2 Zreťazené zásobníky

Zásobník (anglicky stack) je jednou zo základných dátových štruktúr, ktorá sa využíva predovšetkým pre dočasné ukladanie dát v priebehu výpočtu. Zásobník ukladá dáta spôsobom, ktorému sa hovorí *LIFO – last in, first out* – teda posledný vložený prvok smeruje na výstup ako prvý, predposledný ako druhý a tak ďalej.

V ďalšej časti tejto práce sa pracuje s holistickými algoritmami, ktoré používajú práve túto dátovú štruktúru. Pre každý uzol dotazu XPath je vytvorený jeden zásobník. Zásobníky sú potom zreťazené pomocou ukazateľa na rodičovský uzol dotazu. Holistické algoritmy využívajú takúto štruktúru na uchovávanie čiastočných a celkových odpovedí na dotaz XPath. Popíšme si teraz aké funkcie máme k dispozícii pri práci so zásobníkmi:

empty(S_q): signalizuje, či je zásobník S_q prázdny.

pop(S_q): zmaže posledný vložený prvok zo zásobníka S_q.

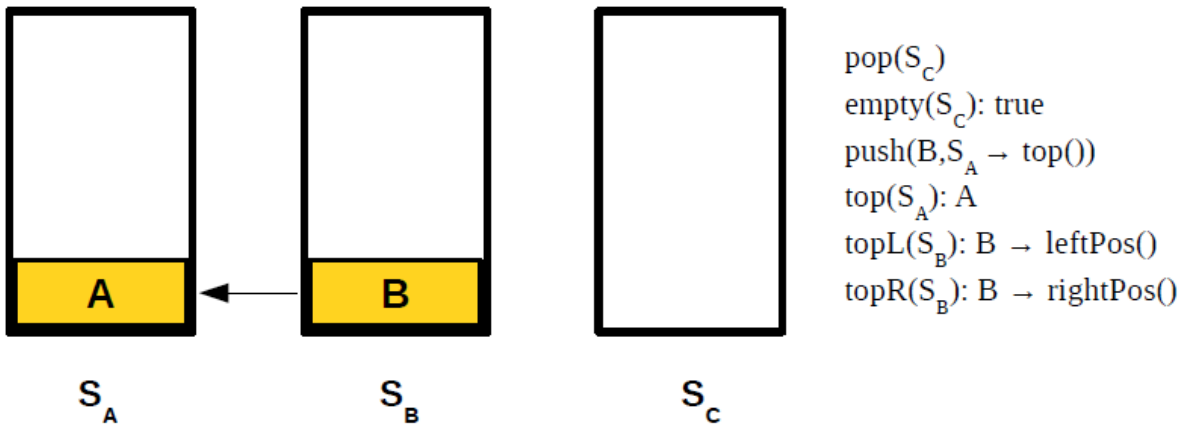
push(S_q, pointer): vloží prvok do zásobníka S_q.

top(S_q): vráti posledný vložený prvok do zásobníka S_q.

topL(S_q): metóda vráti *leftPos* posledného vloženého prvku do zásobníka S_q.

topR(S_q): metóda vráti *rightPos* posledného vloženého prvku do zásobníka S_q.

Všetky spomenuté operácie so zreťazenými zásobníkmi si môžeme prezrieť na obrázku 3.6.



Obrázok 3.6 Operácie so zreťazenými zásobníkmi

3.2 PathStack

Prvý z holistických algoritmov, ktoré si opíšeme je PathStack [8]. Je to algoritmus, ktorý bol navrhnutý na riešenie tzv. *path* dotazov a na vypočítanie odpovedi využíva zreťazené zásobníky. Hlavnou myšlienkou algoritmu PathStack je opakovane konštruovať čiastočné a úplné odpovede na dotaz pomocou zásobníkov, pričom sa opakovane iteruje cez značky, ktoré sú zoradené podľa *leftPos*. PathStack je v algoritme 3.1 znázornený v prípade, že značky pochádzajú iba z jedného dokumentu, avšak je jednoducho rozšíriteľný tak, aby sa pri práci so značkami testovali aj hodnoty *docID*, a rozlišovali sa tak rôzne XML dokumenty. Keďže sa iteruje skrz zoradené značky, odpovede, ktoré sa v zásobníkoch vytvoria sú zoradené od koreňa vetveného dotazu po list vetveného dotazu. Algoritmus končí v momente, kedy sa prejdú všetky značky vo všetkých streamoch. Na začiatku algoritmu, ako nám ukazuje riadok 2, sa vždy vyberie stream s ukazateľom na najnižšiu značku. Na riadkoch 3-5 dochádza ku kontrole zásobníkov, mažú sa vtedy tie značky zo zásobníkov, ktoré nie je možné rozšíriť na konečnú odpoveď podľa vetveného dotazu. Toto mazanie sa vykonáva na základe znalosti ďalšej značky streamu, ktorá sa má spracovať. Inými slovami, mažú sa tie značky, ktoré nie sú predkami aktuálnej značky. Po odstránení všetkých irelevantných značiek vzhľadom ku aktuálnej značke, sa v riadku 6 potom volá procedúra *moveStreamToStack*, ktorá uloží značku do príslušného zásobníka a tiež posunie ukazateľ streamu na ďalšiu značku v poradí. Vždy, keď sa uloží nová značka do zásobníka, ktorý odpovedá listovému uzlu vetveného dotazu, je nájdená nová odpoveď na dotaz, a je zavolaná metóda *showSolutions*.

Algorithm `PathStack`(q)
01. **while** ¬`end`(q)
02. q_{min} = `getMinSource`(q)
03. **for** q_i **in** `subtreeNodes`(q)
04. **while** (¬`empty`(S_{q_i}) ∧ `topR`(S_{q_i}) < `nextL`($T_{q_{min}}$))
05. `pop`(S_{q_i})
06. `moveStreamToStack`($T_{q_{min}}$, $S_{q_{min}}$, pointer to `top`($S_{parent\ (q_{min})}$))
07. **if** (`isLeaf`(q_{min}))
08. `showSolutions`($S_{q_{min}}$, 1)
09. `pop`($S_{q_{min}}$)

Function `end`(q)
return ∀ q_i ∈ `subtreeNodes`(q) : `isLeaf`(q_i) => `eof`(T_{q_i})

Function `getMinSource`(q)
return q_i ∈ `subtreeNodes`(q) : such that `nextL`(T_{q_i}) is minimal

Procedure `moveStreamToStack`(T_q, S_q, p)
01. `push`(S_q, (`next`(T_q), p))
02. `advance`(T_q)

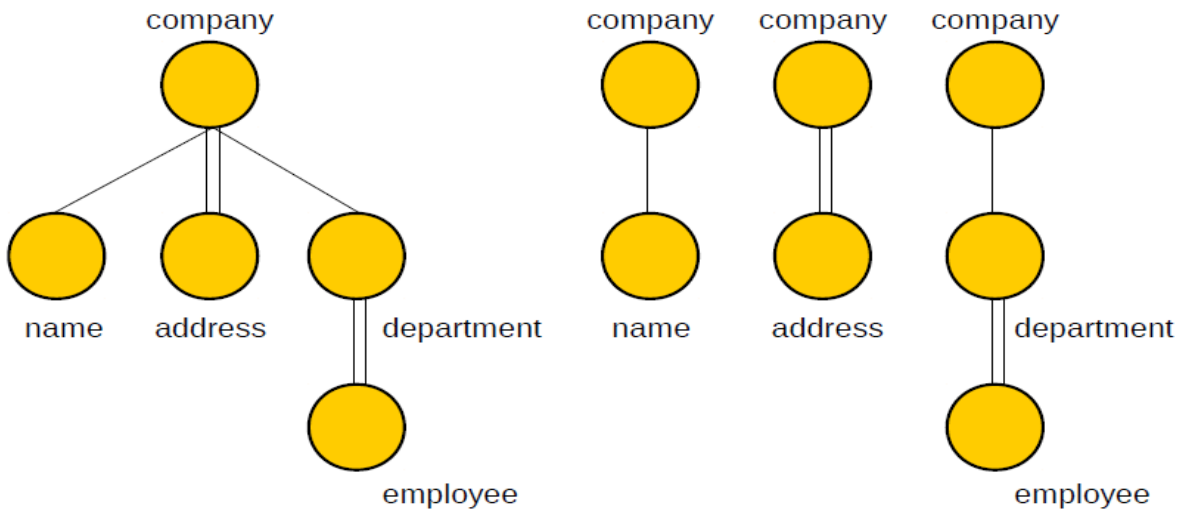
Algoritmus 3.1 PathStack

3.2.1 Zložitosť algoritmu PathStack

Zložitosť algoritmu PathStack je formálne definovaná takto: Majme path dotaz *q* s *n* uzlami, a XML databázu *D*. Algoritmus PathStack má v najhoršom prípade I/O zložitosť a CPU časovú zložitosť lineárnu v súčte veľkostí *n* vstupných a výstupných zoznamov. Ďalej, najhoršia pamäťová zložitosť algoritmu PathStack je minimum zo *(i)* súčet veľkostí *n* vstupných zoznamov, a *(ii)* maximálna dĺžka root-to-leaf cesty v *D*.

3.2.2 Optimalita algoritmu PathStack

Zatiaľ čo pri *path* dotazoch sa ukazuje algoritmus ako veľmi efektívny, horšie to už je pre vetvené dotazy. Vtedy sa musí strom dotazu rozdeliť na niekoľko ciest dotazu (obrázok 3.7), ktoré predstavujú cesty od koreňa po list. Platí teda, že pre každý listový uzol sa vytvorí jedna cesta dotazu. Tento prístup ale čelí rovnakému zásadnému problému, ktoré majú algoritmy s binárnym štrukturálnym spojením - môžu vznikať veľké medzivýsledky, ktoré nebudú súčasťou finálneho riešenia dotazu.



Obrázok 3.7 Rozdelenie vetveného dotazu algoritmom Pathstack

3.3 TwigStack

Keďže použitie PathStack algoritmu pre vetvené dotazy nie je príliš optimálne, bol vytvorený algoritmus TwigStack [8], ktorý je určený na riešenie práve týchto dotazov. Rovnako ako PathStack, aj tento algoritmus používa na výpočet odpovedí reťazce prepojených zásobníkov, ktoré reprezentujú medzivýsledky jednotlivých root-to-leaf (v preklade od koreňa k listu) ciest dotazu. TwigStack (algoritmus 3.2) operuje v dvoch fázach. V prvej fáze (riadky 1-11) sú nájdené odpovede na jednotlivé root-to-leaf cesty dotazu. Každá z týchto odpovedí odpovedá individuálnej ceste dotazu. V druhej fáze algoritmu (riadok 12) sú nájdené čiastočné odpovede spojené do konečných odpovedí odpovedajúcim vetvenému dotazu.

Kľúčovú úlohu v tomto algoritme má pokročilá filtrujúca funkcia *getNext(),* ktorá je volaná v každej iterácii algoritmu TwigStack (riadok 2). Tá zaisťuje, že pred tým ako je uzol

h_q vložený do svojho určeného zásobníka S_q, *(i)* uzol h_q má potomka h_{qi} v každom zo streamov T_{qi}, pre $q_i \in$ children(q) a *(ii)* každý z uzlov h_{qi} rekurzívne spĺňa prvú vlastnosť. Algoritmus PathStack túto vlastnosť nespĺňa, ale ani ju spĺnať nepotrebuje pre *path* dotazy, na ktorý je určený. Funkcia *getNext()* prejde celú stromovú štruktúru vetveného dotazu a skúma značky jednotlivých uzlov vetveného dotazu. Vracia potom tú značku, ktorá vyhovuje poddotazu a má najmenšiu hodnotu *leftPos*. Ak uzol, vrátený funkciou *getNext()* je koreňom dotazu, alebo jeho rodičovský zásobník nie je prázdny, dochádza ku vloženiu značky do príslušného zásobníka. V prípade, že je uzol listom, volá sa funkcia *showSolutionsWithBlocking(S_{Qact},1)*, ktorá pridá novú nájdenú cestu k medzivýsledkom. Celý cyklus prebieha dovtedy, kým sa neprejdú všetky značky v jednotlivých streamoch, ktoré odpovedajú jednotlivým uzlom vetveného dotazu.

Algorithm `TwigStack`(q)
// Phase 1
01. **while** ¬`end(`q`)`
02. q_{act} = `getNext(`q`)`
03. **if** (¬`isRoot(`q_{act}`)`)
04. `cleanStack(parent(`q_{act}`),nextL(`q_{act}`))`
05. **if** (`isRoot(`q_{act}`)` ∨ ¬`empty(`$S_{parent\ (q_{act})}$`)`)
06. `cleanStack(`q_{act}`,next(`q_{act}`))`
07. `moveStreamToStack(`$T_{q_{act}}$, $S_{q_{act}}$, pointer to `top(`$S_{parent\ (q_{act})}$`)`
08. **if** (`isLeaf(`q_{act}`)`)
09. `showSolutionsWithBlocking(`$S_{q_{act}}$`,1)`
10. `pop(`$S_{q_{act}}$`)`
11. **else** `advance(`$T_{q_{act}}$`)`
//Phase 2
12. `mergeAllPathSolutions()`

Function `getNext`(q)
01. **if** (`isLeaf(`q`)`) **return** q
02. **for** q_i **in** `children(`q`)`
03. n_i = `getNext(`q_i`)`
04. **if** ($n_i! \neq q_i$) **return** n_i
05. $n_{\min} = minarg_{n_i}$ `nextL(`T_{n_i}`)`
06. $n_{\max} = maxarg_{n_i}$ `nextL(`T_{n_i}`)`
07. **while** (`nextR(`T_q`)` < `nextL(`$T_{n_{max}}$`)`)
08. `advance(`T_q`)`
09. **if** (`nextL(`T_q`)` < `nextL(`$T_{n_{min}}$`)`) **return** q
10. **else return** n_{min}

Procedure `cleanStack`(S, $actL$)
01. **while** (¬`empty(`S`)` ∧ (`topR(`S`)` < $actL$))
02. `pop(`S`)`

Algoritmus 3.2 TwigStack

3.3.1 Zložitosť algoritmu TwigStack

Zložitosť algoritmu TwigStack je formálne definovaná nasledovne: Majme vetvený dotaz *q*, ktorý je zložený z *n* uzlov, ktoré sú prepojené iba hranami AD, a XML dokument *D*. Algoritmus TwigStack má v najhoršom prípade I/O zložitosť a CPU časovú zložitosť lineárnu so súčtom veľkostí *n* vstupných a výstupných zoznamov. Ďalej, najhoršia pamäťová zložitosť TwigStack je minimum z *(i)* súčet veľkostí *n* vstupných zoznamov, a *(ii)* násobok *n* a maximálnej dĺžky root-to-leaf cesty v *D*.

3.3.2 Optimalita algoritmu TwigStack

Ak zadaný vetvený dotaz obsahuje iba ancestor-descendant hrany, je zaručené, že každá z nájdených ciest, ktorá je uložená v medzivýsledkoch, bude spojená s minimálne jednou ďalšou cestou z medzivýsledku. Z toho vyplýva, že veľkosť medzivýsledkov nebude väčšia ako veľkosť konečného výsledku dotazu. Bohužiaľ sa ale ukázalo, že TwigStack môže vyprodukovať takúto nadbytočnú cestu, ktorá sa nebude dať spojiť so žiadnou inou vyprodukovanou cestou v medzivýsledku, a to v prípade, že sú v dotaze prítomné parent-child hrany. Vtedy už nie je zaručené, že veľkosť medzivýsledkov nebude väčšia ako konečný výsledok dotazu. S tým je spojená suboptimalita časovej a pamäťovej náročnosti algoritmu.

3.4 Nasledovníci algoritmu TwigStack

Od vytvorenia algoritmu TwigStack vzniklo mnoho ďalších algoritmov, ktoré sa snažili inšpirovať týmto algoritmom a ešte viac ho zefektívniť. Jedným z prvých je TwigStackList [9], ktorý sa snaží vylepšiť suboptimalitu PC hrán vetveného dotazu pomocou prídavnej dátovej štruktúry, ktorou je zoznam. Myšlienkou je to, že niektoré uzly čítané z prúdov môžeme ukladať v hlavnej pamäti, vďaka čomu môžeme potom lepšie posúdiť, či sa daný uzol využije v druhej (zlučovacej) fáze algoritmu.

Ďalším algoritmom je iTwigJoin [10]. Tento algoritmus ako prvý dokázal pracovať s *tag+level* a *prefix-path streaming scheme* (kapitola 3.1.1). Tým dokázal eliminovať nadbytočné uzly v streamoch, ktoré mohli vznikať pri použití *tag streaming scheme*. Tým pádom sa celý algoritmus zrýchlil. S použitím týchto nových streamovacích schém vznikol nový termín *stream prunning*. Je to proces, ktorý sa odohráva pred spustením samotného iTwigJoin. V priebehu tohto procesu sa vyberajú relevantné streamy vzhľadom ku vetvenému dotazu za pomoci dátovej štruktúry DataGuide [11], ktorá je modelovaná ako strom a predstavuje základný prístup k indexácií ciest.

Aj algoritmus GTPStack [12] vychádza z algoritmu TwigStack. Tento algoritmus používa kombináciu pokročilého preorder a postorder filtrovania. Kombinovaný prístup poskytuje viaceré výhody:

- Predikátové uzly vetveného dotazu sa nemusia ukladať na zásobníky, čo spôsobuje zrýchlenie vykonania dotazu
- Výrazne znižuje počet uzlov v dočasnom úložisku (intermediate storage) aj v prípade, keď GTPStack nie je optimálny pre daný vetvený dotaz

Algoritmus pracuje s novou pokročilou preorder filtrujúcou funkciou s názvom *getMatch()*. Tá na rozdiel od funkcie *GetNext()* z algoritmu TwigStack zabraňuje nepotrebným volaniam rekurzívnej funkcie a vylepšuje posúvanie streamových kurzorov, čo predstavuje hlavný parameter ovplyvňujúci dobu spracovania pokročilej preorder filtrujúcej funkcie. Zlepšenie dosahuje pomocou troch vylepšení:

- Pridáva mapovanie *Matched:queryNode: bool*, ktoré je implementované ako pole *Matched*, *queryNode* predstavuje uzol vetveného dotazu. *Matched[#queryNode]* vracia *true*, ak sa neposunuli kurzory v streamoch v *subtree(#queryNode)* od posledného volania *getMatch(#queryNode)*.
- Pridáva procedúru *descendantForward()*, ktorá posúva kurzor podľa značky na dne rodičovského zásobníka.
- Predstavuje cyklus, ktorý ukončí volanie funkcie *getMatch()*, až keď sú prečítané všetky uzly v streamoch, alebo bol nájdený uzol $n_{\#q}$, ktorý môže byť koreňom *query match*[9] uzlu *#q* vetveného dotazu.

[9] Query match – je n-tica XML uzlov, kde každý XML uzol odpovedá presne jednému uzlu vetveného dotazu, a vzťahy medzi XML uzlami odpovedajú vzťahom medzi uzlami vo vetvenom dotaze.

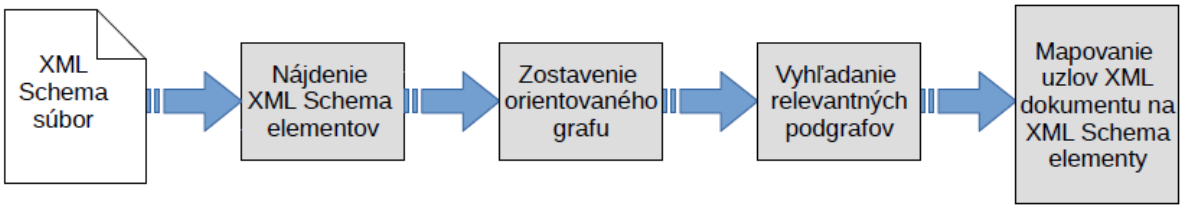
4 Implementácia

4.1 Spracovanie XML Schema

Keďže sa táto práca zaoberá optimalizáciou holistických algoritmov za použitia XML Schema, bolo nutné implementovať mechanizmus, ktorý na vstupe preberie súbor, kde sa nachádza definovaná XML Schema validná ku spracovávanému XML dokumentu, a na výstupe vytvorí jej orientovaný graf. Tento graf je potom vo výsledku tvorený vrcholmi, ktoré predstavujú jednotlivé XML Schema elementy definované v súbore so schémou, a hranami, ktoré predstavujú parent-child vzťahy medzi týmito elementmi. Pri následnom sekvenčnom spracovávaní dokumentu sa snažíme mapovať prichádzajúce uzly z XML dokumentu na príslušné XML Schema elementy v grafe, ktorým XML uzly odpovedajú.

Spomínané mapovanie zaisťuje metóda *Advance*(C_S,U). Táto metóda príjme na vstupe kurzor C_S, ktorý predstavuje aktuálnu pozíciu v grafe XML Schema, pri príchode uzlu U z XML dokumentu. Metóda sa na základe hĺbky vnorenia a názvu uzlu U rozhodne kam kurzor C_S posunúť tak, aby uzol U odpovedal svojmu príslušnému XML Schema elementu v grafe.

Spracovanie XML Schema možno rozdeliť do niekoľkých krokov tak, ako to vidíme na obrázku 4.1. Jednotlivé kroky spracovania XML Schema si detailnejšie rozoberieme v nasledujúcich kapitolách.



Obrázok 4.1 Priebeh spracovania XML Schema

4.1.1 XML Schema Parser

Vytvorený parser pracuje v niekoľkých fázach. V prvej fáze parser prejde celý XML Schema súbor a hľadá výskyty XML Schema elementov, atribútov a užívateľsky definované typy (komplexné typy). V tejto fáze sa ukladajú aj všetky informácie o danom objekte, ktoré sú k dispozícii (názov, typ, referencie na iné objekty a ďalšie dostupné informácie špecifické pre daný XML Schema objekt). V druhej fáze spracovania XML Schema súboru sa vytvárajú definované vzťahy medzi XML Schema elementmi. V poslednej fáze sa potom hľadajú referencie medzi jednotlivými elementmi a upravujú vzťahy medzi nimi. Vyhľadávajú sa aj XML Schema elementy, ktoré majú definovaný vlastný užívateľský typ, ktorý im je priradený, s čím môže byť spojené nájdenie nových potomkov elementu. Tak isto sú nájdené aj prípadne rekurzívne elementy, ktoré môžu byť prítomné. Tie sú potom špeciálne označené aby

v nasledujúcej časti pri vytváraní grafu nevznikol nekonečný cyklus. Na konci parsovania je teda každý element zo schémy reprezentovaný v pamäti. Každý z nich má potom informácie o všetkých svojich deťoch, potomkoch aj predkoch.

4.1.2 XML Schema graf

Inštancia parsera je po ukončení svojej činnosti popísanej v predošlej kapitole predaná inštancii XML Schema grafu. Na začiatku budovania grafu je potrebné určiť, ktorý element je koreňom podľa XML Schema a teda aj koreňom budovaného orientovaného grafu. Keďže inštancia grafu vie, ktorý element je koreňom, môže začať budovať samotný graf od koreňa k listom. Pri jeho budovaní sa spracúva v jeden moment vždy jeden element a využíva sa fakt, že každý element vie o tom, ktoré deti má obsahovať. Danému elementu sú pridané hrany, ktoré smerujú k jeho deťom. Daný postup sa opakuje pre všetky jeho deti. Tento algoritmus sa rekurzívne opakuje až pokiaľ neprejdeme ku všetkým listovým elementom.

Graf je vytváraný tak, že sa pri prítomnosti rekurzívnych elementov v schéme pridá do grafu iba XML Schema element, ktorý začína ďalšie rekurzívne vnorenie, ten už ale neobsahuje žiadne hrany a je listovým uzlom. Na miesto toho, má tento vrchol grafu indikátor, že je rekurzívnym elementom a vlastní ukazateľ na vrchol grafu, na ktorý v schéme odkazuje.

4.1.3 Výber relevantných XML Schema elementov podľa vetveného dotazu

Ako už bolo spomenuté, počas parsovania vstupného XML dokumentu sledujeme, na akej pozícii sa v grafe XML Schema aktuálne nachádzame. Graf XML Schema si môžeme pripraviť na základe vetveného dotazu tak, aby sa pri parsovaní XML dokumentu spracovávali iba tie uzly XML dokumentu, ktoré odpovedajú relevantným XML Schema elementom.

Pre každý uzol vetveného dotazu je vytvorená štruktúra *TPQ_SchemaElement_mapping* (algoritmus 4.1), ktorá je následne vložená do poľa *Mappings*. Štruktúra disponuje informáciami o svojom uzle vetveného dotazu, ďalej obsahuje odkaz (mapovanie) na XML Schema element, ktorý je inicializovaný ako *NULL*, a obsahuje aj indikátor *recursiveSearch* inicializovaný na hodnotu *false*, ktorý bráni vzniku nekonečných cyklov pri prehľadávaní rekurzívnych XML Schema elementov.

```
//Struktura mapuje XML Schema elementy na uzly vetveneho dotazu
struct TPQ_SchemaElement_mapping{
   TPQNode node;
   SchemaElement element;
   bool recursiveSearch;
};
```

Algoritmus 4.1 Definícia mapovacej štuktúry TPQ_SchemaElement_mapping

Algoritmus 4.2 znázorňuje, ako sa pre XML Schema elementy relevantnosť určuje. V prvej fáze tohto algoritmu sa vykonáva metóda *CollectElements()*, ktorá má za úlohu prejsť všetky XML Schema elementy, pričom si do pripravenej kolekcie uloží tie, ktoré sa v názve zhodujú s koreňovým uzlom vetveného dotazu.

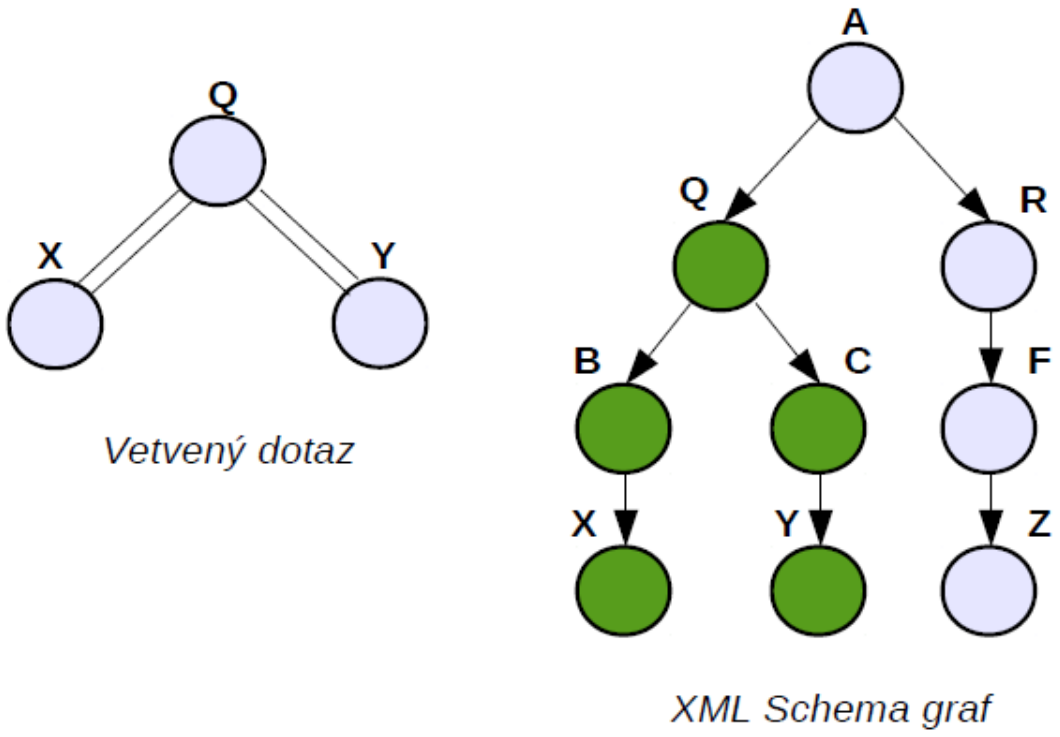
```
void PreprocessXmlSchemaGraphAccordingTPQ()
{
   CollectElements(SchemaGraphRootElement,RootTPQNode);

   forall element ∈ collectedElements do
   {
      ControlElementTwigs(RootTPQNode,element);

      if vsetky uzly vetveneho dotazu maju priradeneho jednoho z potomkov
        element then
      {
          nastav relevanciu pre element;
      }

      Resetuj hodnoty pre vsetky mapovacie instancie;
   }

   forall element ∈ relevantElements do
   {
      SchemaElement lastEl = Najdi pre element posledneho pre-order potomka;

      Nastav vlastnost pre element nasledujuci v pre-order po lastEl ktora
     znaci ukoncenie relevancie;
   }
}
```

Algoritmus 4.2 Pseudokód algoritmu, ktorý pripravuje relevantné XML Schema elementy na parsovanie podľa vetveného dotazu

V druhej fáze algoritmu 4.2 potom testujeme každý z uložených XML Schema elementov pomocou metódy *ControlElementTwigs()*. Jej úlohou je zistiť, či je možné aktuálne skúmaný XML Schema element rozšíriť do podoby vetveného dotazu. Po preskúmaní rozšíriteľnosti aktuálne skúmaného XML Schema elementu nasleduje kontrola, či sa každej štruktúre v poli *Mappings* podarilo namapovať svoj uzol vetveného dotazu na XML Schema element – vtedy je zrejmé, že aktuálne skúmaný XML Schema element sa podarilo rozšíriť do podoby vetveného dotazu a nastavíme relevanciu pre aktuálny XML Schema element. Pred skúmaním ďalšieho z uložených XML Schema elementov ešte upravíme hodnoty inštancií v poli *Mappings* na počiatočné hodnoty. Po preskúmaní každého z uložených XML Schema elementov potrebujeme ešte upraviť podgraf každého relevantného XML Schema elementu. Keďže optimalizácia funguje tak, že sa spracúvajú iba uzly XML dokumentu, ktoré odpovedajú potomkom (podgrafu) relevantného XML Schema elementu, pre každý relevantný XML Schema element preto vyhľadáme jeho posledného potomka *lastEl* v pre-order prechádzaní grafu. Tento XML Schema element bude posledným, ktorý sa bude spracovávať vďaka relevancii predka. XML Schema element, ktorý nasleduje v pre-order poradí po *lastEl* je potom špeciálne označený a filtrácii oznamuje, že už sa nevyskytujú relevantné uzly.

Príklad si môžeme prezrieť na obrázku 4.2, kde máme zadaný vetvený dotaz a XML Schema graf. Keď sa v priebehu parsovania XML dokumentu narazí na XML uzol Q, ktorý odpovedá XML Schema elementu Q (mapovaním odpovedá koreňovému uzlu vetveného dotazu Q_D), filtrácii sa predá informácia, ktorá jej oznamuje, že aktuálne sa môžu vyskytovať relevantné XML uzly. V tomto prípade sa teda budú spracovávať všetky uzly XML dokumentu, ktoré odpovedajú XML Schema elementom Q, B, X, C a Y. Posledným pre-order po-

tomkom XML Schema elementu Q je Y. Po ňom v pre-order prechádzaní nasleduje XML Schema element R, ktorý je špeciálne označený tak, že filtrácii hovorí, že už sa nevyskytujú relevantné uzly XML dokumentu.



Obrázok 4.2 Vyznačenie spracovávaných XML Schema elementov pre daný vetvený dotaz

Metóda *ControlElementTwigs* (algoritmus 4.3) je založená na súčasnom rekurzívnom prechádzaní stromu vetveného dotazu a grafu XML Schema. Prebieha v nej samotné mapovanie XML Schema elementov na uzly vetveného dotazu. V momente, kedy sú na vstupe uzol vetveného dotazu U_Aa XML Schema element E_A s rovnakým názvom vykonávame nasledovné:

- majme uzol vetveného dotazu U_{rodicA} , ktorý je rodičovským uzlom aktuálne mapovaného uzlu vetveného dotazu U_A. V momente kedy prehľadávame XML Schema graf, a hľadáme XML Schema element k U_A, musia byť splnené vzťahy PC resp. AD medzi aktuálnym XML Schema elementom E_A a XML Schema elementom E_{rodicA}, ktorý je uložený v inštancii štruktúry *TPQ_SchemaElement_mapping*, ktorá odpovedá uzlu vetveného dotazu U_{rodicA}.

Ak je daná podmienka splnená, je nájdený XML Schema element, ktorý by mohol odpovedať tomuto uzlu vetveného dotazu a vykoná sa uloženie (mapovanie) *elementu* do tej inštancie štruktúry, ktorá má odkaz na práve tento *uzol* vetveného dotazu. Ak je na vstupe tejto metódy koreňový uzol vetveného dotazu (nastane v prípade prvého volania metódy z algoritmu 4.2), tak je XML Schema element bez ďalšej kontroly (nemá predkov) uložený do odpovedajúcej inštancie štruktúry.

```
void ControlElementTwigs(TPQNode node,SchemaElement element)
{
   if (element je rekurzivny && recursiveSearch pre element ma hodnotu false) then
   {
      Nastav pre element vlastnost recursiveSearch na true;
      ControlElementTwigs(node,odkazovany element);
   }

   if node je koren dotazu then
   {
      Uloz mapovanie dvojce node a element;

      forall childNode ∈ node do
      {
         forall childElement ∈ element do
         {
            ControlElementTwigs(childNode,childElement);
         }
      }
   }
   else
   {
      typHrany = Zisti typ hrany od node ku svojmu rodicovi;

      switch(typHrany)
      {
         case AD:
         {
            if nazov node a nazov elementu sa zhoduju then
            {
               Uloz mapovanie dvojce node a element;

               forall childNode ∈ node do
               {
                  forall childElement ∈ element do
                  {
                     ControlElementTwigs(childNode,childElement);
                  }
               }
            }
            else
            {
               forall childElement ∈ element do
               {
                  ControlElementTwigs(node,childElement);
               }
            }
         }
         break;

         case PC: ...
      }
   }
}
```

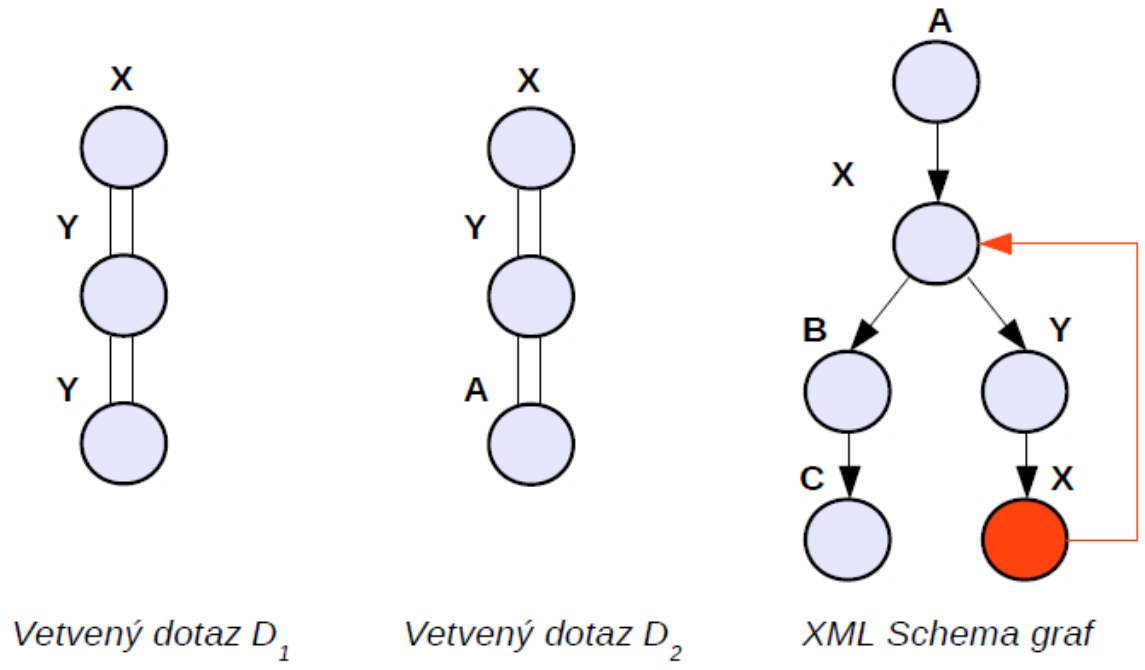
Algoritmus 4.3 Pseudokód rekurzívnej metódy, ktorá mapuje XML Schema elementy na uzly vetveného dotazu. Metóda je využitá v algoritme 4.2

Ak na vstupe nie je koreňový uzol vetveného dotazu, zisťujeme typ hrany medzi aktuálnym uzlom vetveného dotazu (pre ktorý sa snažíme nájsť XML Schema element) a rodičovským uzlom. Potom postupujeme podľa odpovedajúceho vzťahu PC resp. AD. Pri vzťahu PC sa

navyše kontroluje hĺbka porovnávaných XML Schema elementov – platí, že hĺbka elementu E_1 musí byť presne o 1 väčšia ako elementu E_2, ktorý má byť jeho rodičom.

Metóda *ControlElementTwigs* si dokáže poradiť aj s rekurzívnym vyhľadávaním XML Schema elementov. V každom volaní sa kontroluje či aktuálny XML Schema element nie je rekurzívny – teda či neobsahuje referenciu na iný element. Ak je tomu tak, zmení sa indikátor *recursiveSearch* odpovedajúcej štruktúry *TPQ_SchemaElement_mapping* na hodnotu *true*. V prípade, že nájdeme hľadaný XML Schema element pred opätovným navštívením rekurzívneho XML Schema elementu, hodnotu *recursiveSearch* zmeníme na *false*, aby sme umožnili hľadanie v ďalšom rekurzívnom vnorení elementov. Naopak ak predtým ako sa znovu dostaneme k rekurzívnemu XML Schema elementu nenájdeme odpovedajúci XML Schema element pre aktuálne mapovaný uzol dotazu, hodnota *recursiveSearch* ostáva *true*. Potom, keď príde na vstup znova XML Schema element, ktorý je rekurzívny, neumožníme už jeho posunutie na odkazovaný XML Schema element. Rekurzívny XML Schema element už nemá žiadnych potomkov v grafe, a tak je dané volanie metódy *ControlElementTwigs* ukončené.

Pre lepšiu orientáciu sa pozrime na obrázok 4.3, kde je zobrazený XML Schema graf s rekurzívnym XML Schema elementom X (označený červenou farbou), ktorý má ukazateľ na odkazovaný XML Schema element.

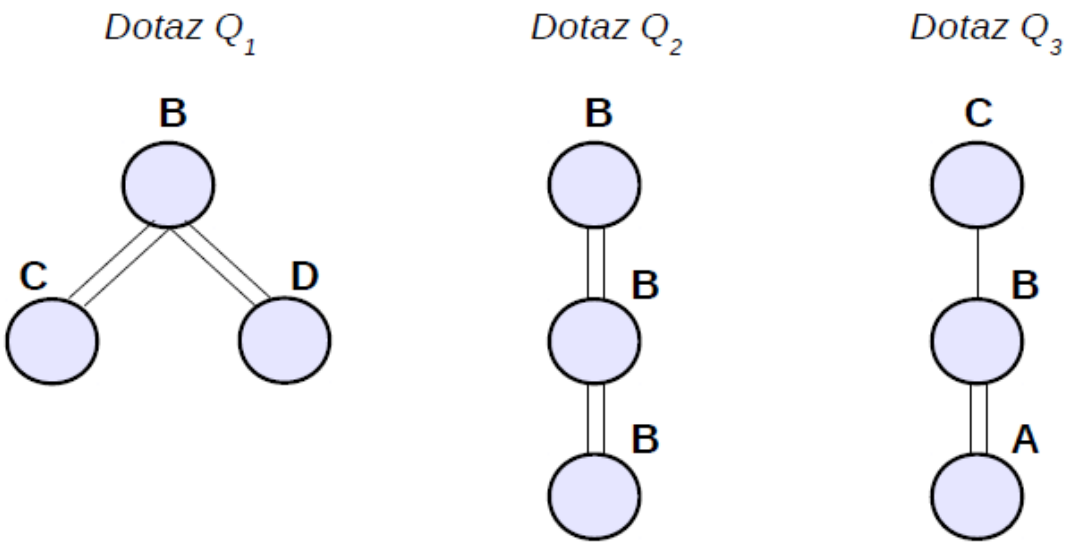


Obrázok 4.3 Vykonanie rekurzívnych dotazov na grafe XML Schema

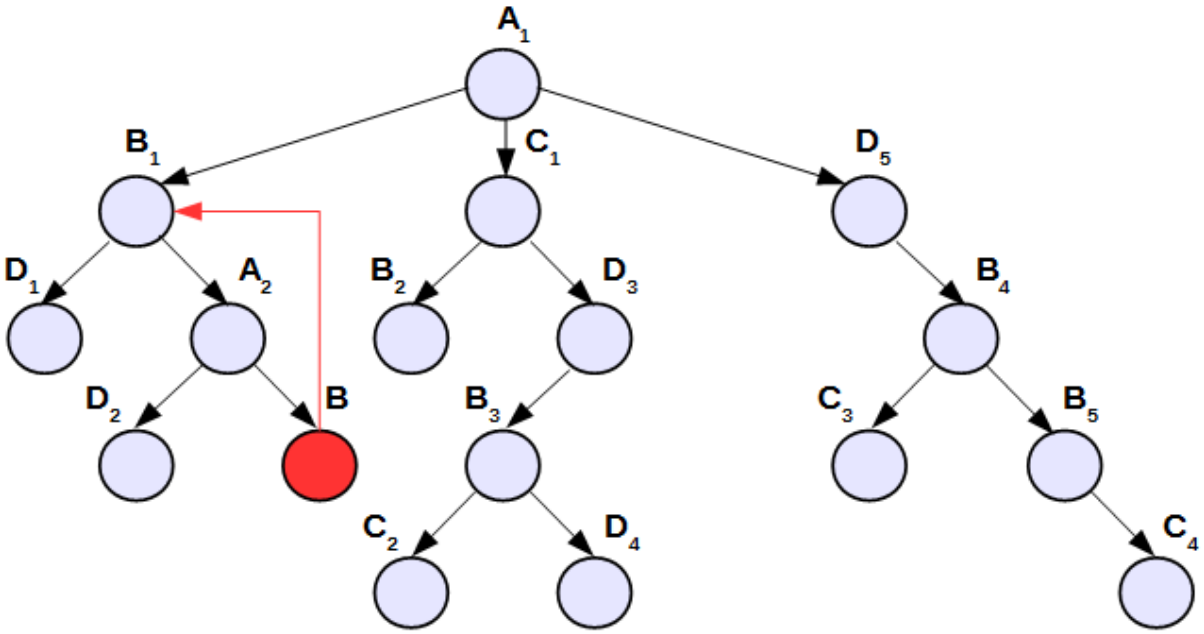
Keď si vezmeme dotaz D_1, metóda *ControlElementTwigs* bude mať na vstupe uzol vetveného dotazu X_D a XML Schema element X. Postupne pri ďalšom prechádzaní XML Schema grafu metóda narazí na XML Schema element Y a prebehne mapovanie ku uzlu vetveného dotazu Y_D. V tomto momente treba nájsť ďalší XML Schema element Y, ktorý by odpovedal listovému uzlu vetveného dotazu. Pri ďalšom prechádzaní XML Schema grafu sa dostaneme k listovému rekurzívnemu XML Schema elementu X. Keďže má vtedy inštancia štruktúry, ktorá ma na starosti mapovanie listového uzlu Y vetveného dotazu, hodnotu *recursiveSearch*

= *false*, môžeme sa posunúť na odkazovaný XML Schema element a dohľadať XML Schema element Y, vďaka čomu sa určí relevancia vstupného XML Schema elementu X. Keď si zoberieme dotaz D_2, postup metódy bude spočiatku rovnaký, postupne prebehne mapovanie uzlov X_D, Y_D vetveného dotazu na XML Schema elementy X,Y. V tomto momente máme ešte vyhľadať XML Schema element, ktorý by odpovedal listovému uzlu A_D vetveného dotazu. Opäť sa postupne posunieme z listového rekurzívneho XML Schema elementu X na odkazovaný XML Schema element X a zmeníme hodnotu *recursiveSearch* na *true* tej inštancii štruktúry, ktorá mapuje XML Schema elementy na uzol vetveného dotazu A_D. Pri následnom prechádzaní rekurzívneho vnorenia XML Schema elementov v tomto podgrafe ale nenájdeme žiadny odpovedajúci hľadaný XML Schema element A, a v momente, kedy sa pri prechádzaní XML Schema grafu dostaneme znova ku rekurzívnemu XML Schema elementu X, už je pre odpovedajúcu inštanciu štruktúry nastavená hodnota *recursiveSearch* = *true.* Nedovolíme teda ďalšie posunutie aktuálnej pozície v grafe XML Schema na odkazovaný XML Schema element a zistíme, že pre daný dotaz D_2 neexistuje relevantný podgraf. To znamená, že pre tento dotaz neexistuje v grafe XML Schema relevantný XML Schema element.

Na obrázku 4.4 sú príklady vetvených dotazov, ktoré uplatníme na XML Schema graf (s rekurzívnym elementom B červenej farby) z obrázku 4.5. Vyššie popisovaná optimalizácia nám zaisťuje, že podľa dotazu Q_1 nemusíme spracovávať uzly, ktoré odpovedajú XML Schema elementom B_1, B_2, B_4 a B_5, pretože nie sú relevantné podľa daného dotazu. Podľa dotazu Q_2 stačí ak budeme spracovávať uzly, ktoré sú potomkami B_1(vnorený rekurzívny XML uzol B môže mať ďalších potomkov D). Na koniec na základe dotazu Q_3 môžeme už pred spracovávaním XML dokumentu zistiť, že žiadne relevantné uzly dokument obsahovať nebude.

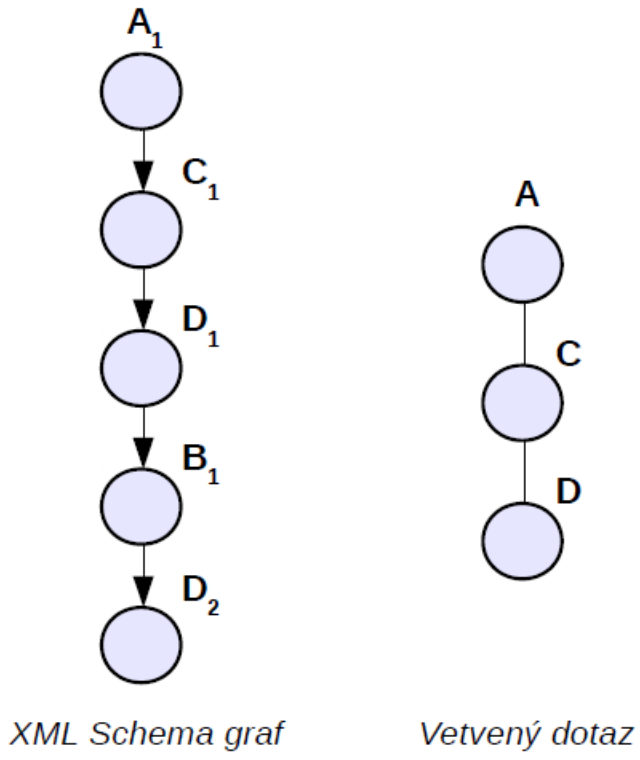


Obrázok 4.4 Dotazy ku XML Schema grafu z obrázku 4.5



Obrázok 4.5 XML Schema graf s rekurzívnym XML Schema elementom (označený červenou farbou)

Ako už bolo spomenuté, optimalizácia funguje tak, že sa spracúvajú iba uzly XML dokumentu, ktoré odpovedajú potomkom relevantného XML Schema elementu. Avšak ani to neznamená, že všetky XML Schema elementy v danom podgrafe musia byť relevantné pre vetvený dotaz. Túto skutočnosť máme možnosť vidieť na obrázku 4.6.



Obrázok 4.6 Situácia, kedy nie všetky elementy v podstrome A_1 sú pre dotaz relevantné

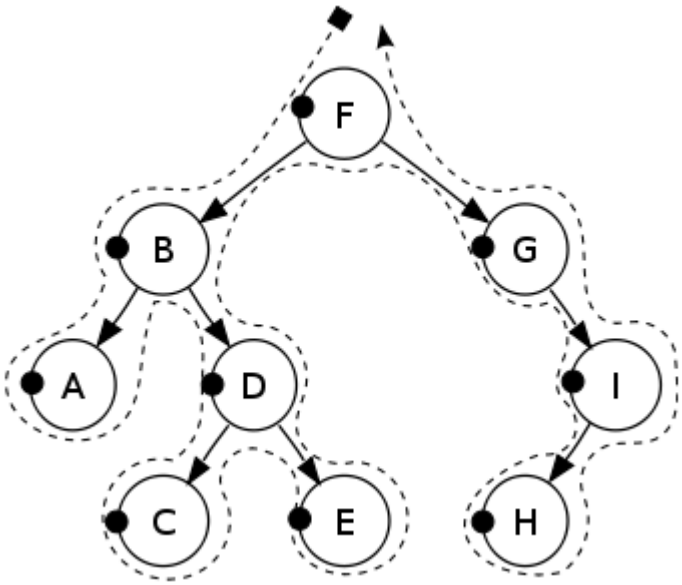
XML Schema element D_2 nie je relevantný, pretože nemá rodiča s názvom C. Preto každý XML Schema element, ktorý bol v začiatočnej fáze uložený do medzipamäte ako relevantný, sa ďalej skúma jeho podgraf. XML Schema Elementy v podgrafe, ktoré nemajú odpovedajúceho predka (rodiča) sú označené vlastnosťou, ktorá signalizuje, že nemajú byť spracovávané, pretože nie sú pre vetvený dotaz relevantné.

Táto optimalizácia nám teda umožňuje na základe XML Schema vynechávať uzly XML dokumentu, ktoré nebudú relevantné podľa vetvenému dotazu, s čím je spojená nižšia časová aj pamäťová zložitosť algoritmu.

4.1.4 Mapovanie uzlov XML dokumentu na XML Schema elementy

XML Schema určuje, ako má vypadať daný XML dokument ku nej validný. Keď si predstavíme oba zobrazené v stromovej štruktúre, zistíme, že aby sme sledovali v grafe schémy pohyb pri spracovávaní uzlov XML dokumentu, musíme sledovať stromovú štruktúru XML Schema tzv. Pre-order prechádzaním grafu. Tento spôsob prechádzania grafu sa skladá z troch krokov [13] (obrázok 4.7):

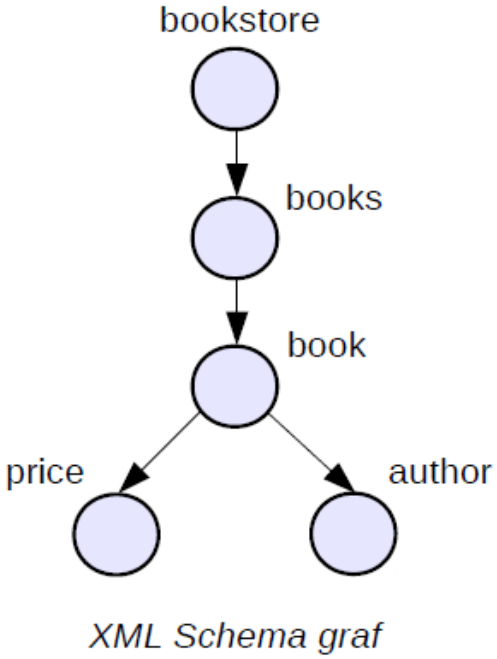
1) Navštív koreň
2) Prejdi ľavý podstrom grafu
3) Prejdi pravý podstrom grafu



Obrázok 4.7 Pre-order prechádzanie grafu – F, B, A, D, C, E, G, I, H

Každý XML Schema element z vytvoreného grafu má teda nastavený ukazateľ na XML Schema element, ktorý nasleduje po ňom v spomínanom pre-order prechádzaní grafu.
Zvyčajne sa v XML dokumente nachádza viacero uzlov, ktoré majú rovnakého rodiča a odpovedajú rovnakému XML Schema elementu. To môžeme sledovať na obrázku 4.8, kde XML Schema element *books*, obsahuje element *book*. V XML dokumente sa môže vyskyto-

vať niekoľko uzlov *book*, ktoré sú v XML Schema príslušné rovnakému XML Schema elementu *book*, majú pri tom rovnakého rodiča *books*.

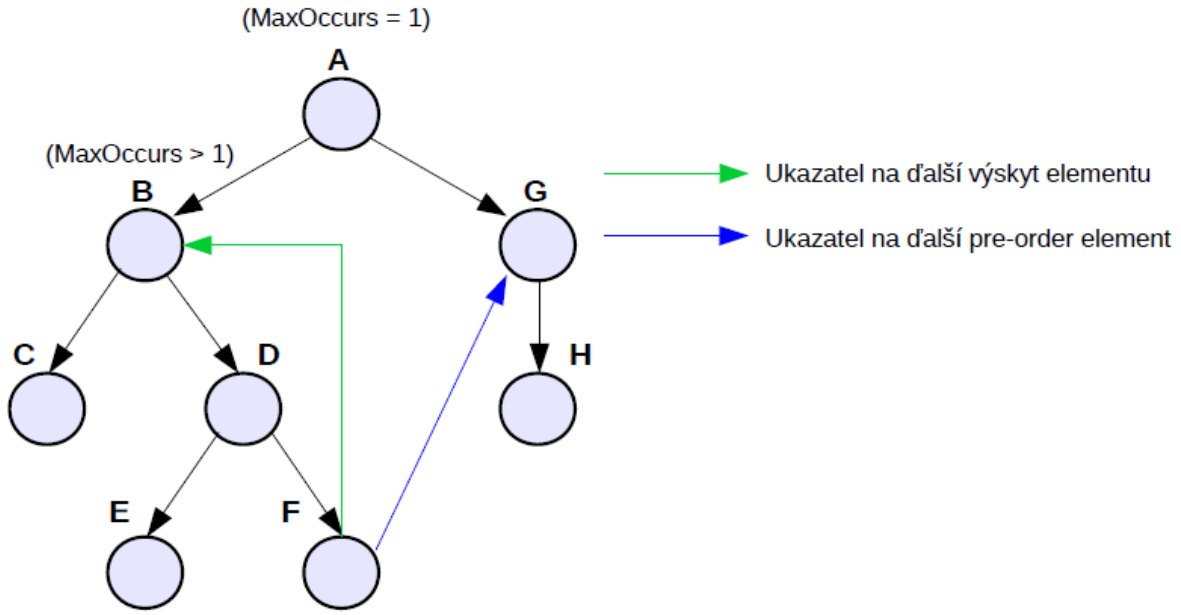



```
<?xml version="1.0"?>
<bookstore>
  <books>
    <book>
      <price>35</price>
      <author>Coolman</author>
    </book>
    <book>
      <price>50</price>
      <author>O'Neill</author>
    </book>
    <book>
      <price>47</price>
      <author>Brown</author>
    </book>
    <book>
      <price>77</price>
      <author>Davis</author>
    </book>
  </books>
</bookstore>
```

Obrázok 4.8 Viacnásobný výskyt uzlov XML dokumentu (book), ktoré majú rovnakého rodiča (books), a odpovedajú rovnakému XML Schema elementu

Ako bolo popísané v kapitole 1.4.2.2, iný počet výskytov elementu ako je 1 sa v XML Schema deklaruje pomocou vlastností *minOccurs* a *maxOccurs*. Vtedy nastane situácia, kedy nasledujúci vrchol v grafe nemusí byť ten v pre-order poradí, a je nutné sa rozhodnúť kam posunúť aktuálny ukazateľ v grafe XML Schema.

Pozrime sa na obrázok 4.9, kde XML Schema element *B* má definovaný výskytový atribút *maxOccurs* vyšší ako 1 a jeho rodič XML Schema element má definovaný výskytový atribút *maxOccurs* rovný 1 (u všetkých ostatných XML Schema elementov predpokladáme *maxOccurs*=*minOccurs* = 1). V tomto prípade, keď budeme mapovať prichádzajúce uzly XML dokumentu na XML Schema elementy, v určitom čase dospejeme k situáciu, kedy máme na vstupe aktuálny kurzor XML Schema nastavený na XML Schema element *F*, a nový spracovaný uzol *U*. XML Schema element *F* je posledným potomkom vrcholu *B*. V tejto situácii môže uzol *U* byť buď uzol s názvom G, ktorý by v grafe odpovedal ďalšiemu XML Schema elementu v pre-order prechádzaní grafu, alebo to môže byť uzol s názvom B, ktorý odpovedá pri mapovaní ďalšiemu výskytu XML Schema elementu *B*. Preto, každý XML Schema element, ktorý končí výskyt podgrafu XML Schema elementu s vyšším počtom výskytov ako je 1, má nastavený aj ukazateľ na XML Schema element, ktorý začína ďalší výskyt. Podľa aktuálneho uzlu, ktorý príde na vstup pri parsovaní sa potom rozhodujeme (podľa jeho mena, a hĺbky zanorenia v XML), kam ukazateľ na XML Schema posunúť.

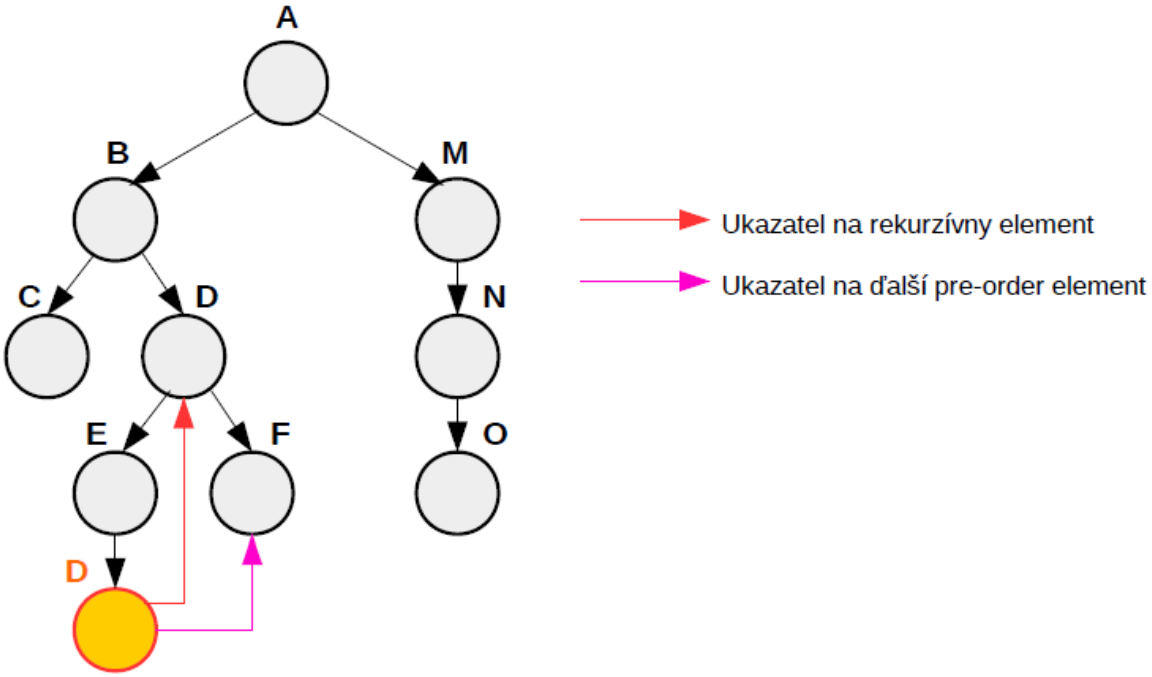


Obrázok 4.9 Mapovanie XML Schema elementov pri nastavení atribútu maxOccurs > 1

Potenciálny problém by mohla predstavovať situácia, kedy by sme v obrázku 4.9 nahradili XML Schema element *G*, elementom s názvom *B*. Vtedy keď by sme pri mapovaní dospeli k XML Schema elementu *F*, by sme mali na výber buď element *B* s hĺbkou 2, ktorý by symbolizoval ďalší výskyt predka vrcholu *F*, alebo druhú možnosť by predstavoval opäť element s názvom *B* a s hĺbkou 2, ktorý by nasledoval po vrchole *F* pri pre-order prechádzaní grafu. Lenže podľa definície XML Schema nemôže táto situácia nastať, pretože podľa nej môžeme deklarovať dva súrodenecké elementy s rovnakým názvom iba vtedy, ak majú rovnaký typ (čo zahŕňa aj rovnakých potomkov v rovnakom poradí). Vtedy je teda zbytočné deklarovať rovnaký XML Schema element niekoľko krát - stačí ak mu nastavíme už spomínanú *maxOccurs* vlastnosť.

Podobný postup je potom použitý aj pri mapovaní rekurzívne vnorených uzlov v XML dokumente na XML Schema elementy. Pozrime si tento krát obrázok 4.10. Farebne označený XML Schema element *D* je rekurzívny. V XML dokumente validnom podľa tejto schémy to znamená, že každý XML uzol E môže ďalej obsahovať potomka D, a ten následne potomkov E a F. Každý rekurzívny XML Schema element je špeciálne označený a okrem ukazateľa na ďalší pre-order XML Schema element obsahuje aj ukazateľ na XML Schema element, na ktorý v schéme odkazuje – tu posunieme kurzor XML Schema grafu v prípade, že na vstup príde ďalší výskyt rekurzívne vnoreného XML uzlu D, ktorý má rodiča E. V prípade, že na vstup príde uzol XML dokumentu F, uzol XML dokumentu E už neobsahuje ďalšie rekurzívne vnorenie uzlu XML dokumentu D, a kurzor XML Schema posúvame na ďalší pre-order XML Schema element *F*. Aj toto rozhodovanie je učinené na základe názvu prichádzajúceho uzlu a jeho hĺbky. Navyše sa pri tejto činnosti používa pomocné počítadlo, ktoré signalizuje počet vnorení rekurzívnych XML uzlov so spoločným predkom, ktoré odpovedajú práve rekurzívnemu XML Schema elementu. V momente, kedy sa kurzor XML Schema grafu pohne z rekurzívneho XML Schema elementu na ďalší pre-order XML Schema element (z rekurzívneho elementu *D* na element *F*) začneme sledovať toto počítadlo, ktoré určuje koľko krát

sa budú opakovať XML uzly odpovedajúce v grafe XML Schema potomkom rekurzívneho XML Schema elementu, ktoré ešte neprišli na vstup (potomok F rekurzívneho uzlu D).



Obrázok 4.10 Mapovanie rekurzívnych XML Schema elementov

Pri rozhodovaní nemusí názov uzlov XML dokumentu hrať žiadnu úlohu, pretože názvy XML Schema elementov medzi ktorými sa rozhodujeme môžu byť rovnaké. Avšak v tom prípade využívame hĺbku vnorenia aktuálneho XML uzlu, ktorá nám prezradí, ktorému XML Schema elementu aktuálny uzol odpovedá.

Na koniec je potrebné poznamenať, že XML Schema element *choice* (kapitola 1.4.2.2) môže byť deklarovaný v schéme rozličnou zložitou formou. V niektorých situáciách by bolo iba veľmi obtiažne mapovať uzly XML dokumentu v XML Schema orientovanom grafe.

4.2 Filtrácia uzlov XML Dokumentu

Ako už bolo vysvetlené v kapitole 3.1.1, holistické algoritmy pracujú so streamami uzlov XML dokumentu, ktoré sú priradené každému uzlu vetveného dotazu. Účelom tejto práce je práve redukcia veľkosti týchto streamov, ktoré sú vstupom pre holistický algoritmus, s čím by mohla byť spojená určitá časová a pamäťová optimalizácia behu algoritmu. Redukcia je založená na skutočnosti, že streamy často obsahujú značky, ktoré sú pre daný dotaz irelevantné. Táto optimalizácia nie je závislá na konkrétnom použitom holistickom algoritme, je teda možné ju použiť pre všetky holistické algoritmy, ktoré sú na vstupe schopné pracovať s *tag streaming scheme*. V práci boli vykonané dva typy optimalizácie: *prefixová* optimalizácia a *postfixová* optimalizácia. Redukcia sa vykonáva v priebehu sekvenčného spracovávania XML dokumentu. Už bolo spomenuté v predošlej kapitole, že počas spracovávania dokumen-

tu sa v závislosti na prichádzajúcej značke sleduje ukazatel na aktuálnu pozíciu v grafe XML Schema. To nám umožňuje rozhodovať, či má cenu pracovať s aktuálnou značkou (nemusí byť relevantná). Pracujeme s ňou teda iba vtedy, ak kurzor na graf XML Schema odkazuje na XML Schema element, ktorý bol označený ako relevantný ešte pred spracovávaním XML dokumentu. V ďalšej časti tejto práce používame na miesto pojmu uzol XML dokumentu termín značka, ktorá odpovedá *Element Labeling Scheme* (kapitola 2.2).

4.2.1 Prefixová optimalizácia

Prefixová optimalizácia sa zameriava na uzly vetveného dotazu, ktoré sú listami[10]. Optimalizácia prebieha na základe prefixu listového uzlu vetveného dotazu U_{list}, pričom prefix predstavuje množinu uzlov vetveného dotazu, ktoré sú predkami práve uzlu U_{list}. V praxi to znamená, že kontrolujeme, či daný listový uzol vetveného dotazu U_{list} má odpovedajúcich predkov podľa svojho prefixu, a až v prípade, že je tomu tak, je značka pridaná do streamu U_{list}. Samotná filtrácia uzlov je založená na algoritme PathStack (kapitola 3.2) bez využitia enumerácie jednotlivých značiek v zásobníkoch. V momente, kedy je vložená značka do zásobníka listového uzlu dotazu, je nájdená značka, ktorá je podľa dotazu relevantná, a je zavolaná procedúra, ktorá zabezpečí jej uloženie do streamu značiek listového uzlu dotazu. Ostatné streamy sú plnené rovnakým spôsobom – v momente, kedy je značka uložená na príslušný zásobník, je uložená aj do príslušného streamu uzlu vetveného dotazu.

Predstavme si, že vykonáme dotaz $D//B_i//B_{ii}$ na XML dokumente so schémou na obrázku 4.11 (červený XML Schema element je rekurzívny). Ak by sme použili tag streaming scheme bez optimalizácie, boli by do streamu uzlu B_{ii} vložené všetky značky (XML uzly), ktoré majú v dokumente XML názov B. Boli by to teda všetky značky, ktoré odpovedajú XML Schema elementom B_1, B_2, B_3, B_4, B_5. Prefixová optimalizácia pri spracovaní dokumentu nám zaistí, že do streamu uzlu dotazu B_{ii} budú vložené iba tie značky, ktoré odpovedajú XML Schema elementu B_5, pretože iba tento XML Schema element má odpovedajúci prefix – predkov.

4.2.2 Postfixová optimalizácia

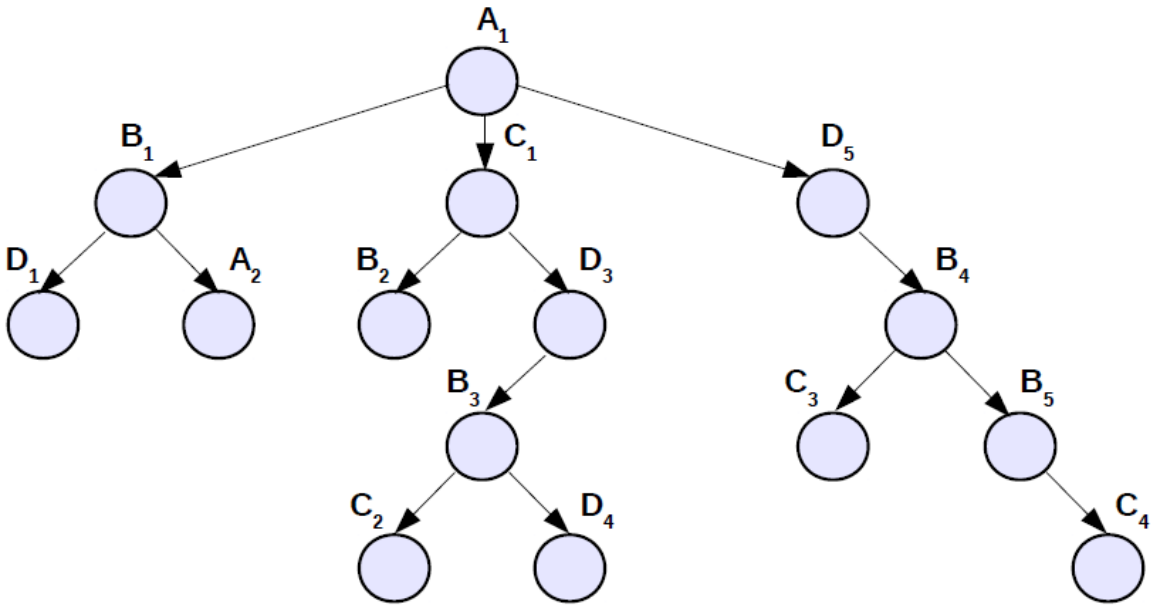
Táto technika optimalizácie funguje obrátene v porovnaní s prechádzajúcou prefixovou. Zatiaľ čo prefixová optimalizácia redukovala značky príslušné streamu listového uzlu vetveného dotazu a všímala si aké má mať listový uzol vetveného dotazu predkov, postfixová optimalizácia redukuje značky pre koreňový uzol U_{koren} vetveného dotazu a všíma si, akých má mať U_{koren} potomkov. Znamená to teda, že predtým ako vložíme do streamu uzlu U_{koren} danú značku A_1, overíme, či značka A_1 má odpovedajúcich potomkov podľa dotazu daného vetveného dotazu. Aj táto optimalizácia používa na filtrovanie upravený algoritmus PathStack (kapitola 3.2) bez enumerácie značiek v zásobníkoch. Do výsledného streamu uzlu U_{koren} môžeme pridať značku zo zásobníka S_{koren} v tom momente, keď budú všetky zásobníky obsahovať aspoň jednu značku. Vtedy zároveň vyprázdnime všetky zásobníky, ktoré odpovedajú listovému uzlu vetveného dotazu. Pri vkladaní uzlu do streamu U_{koren} musíme

[10] Listové uzly neobsahujú už žiadnych ďalších potomkov

ďalej kontrolovať, či už tam táto značka nebola vložená. O to sa stará funkcia, ktorá používa binárne vyhľadávanie, ktoré je možné použiť vďaka tomu, že XML dokument spracovávame sekvenčne a jednotlivé značky sú zoradené v streame v tomto prípade podľa svojej *leftPos*.

Keďže táto optimalizácia pracuje rovnakým princípom ako prefixová (ukladá značku na daný zásobník vtedy ak má na rodičovskom zásobníku prítomného predka), aj táto optimalizácia filtruje prefixovo, čo bude možné vidieť pri výsledkoch testovania v nasledujúcej kapitole.

Predstavme si, že vykonáme dotaz *C[//D]//B* na XML dokumente so schémou na obrázku 4.11. Ak by sme použili tag streaming scheme bez optimalizácie, pre koreňový uzol vetveného dotazu *C* by boli vložené do streamu všetky značky v XML dokumente s menom C, ktoré odpovedajú XML Schema elementom C_1, C_2, C_3, C_4. Za pomoci postfixovej optimalizácie ale dosiahneme, že do streamu pre uzol vetveného dotazu *C* budú vložené iba značky odpovedajúce XML Schema elementu C_1, pretože iba on môže mať odpovedajúci postfix – potomkov.



Obrázok 4.11 Príklad XML Schema grafu

5 Testovanie

5.1 Testovacie dáta

Vytvorenú aplikáciu sme otestovali na reálnych XML dátach. Testované dokumenty sú:

- *enwiki-latest-stub-meta-current26.xml*[11] – prvý testovaný dokument predstavuje zálohu časti dát z Wikipedie o veľkosti 1,4GB. Celkovo obsahuje 40 701 115 uzlov.
- *proteinDb.xml*[12] - je to voľne dostupná databáza skúmaných proteínov. Veľkosť má 700MB a obsahuje 21 305 818 uzlov.
- *recursiveDocument.xml* – ide o manuálne vytvorený rekurzívny XML dokument s veľkou hĺbkou vnorenia rekurzívnych uzlov. Má najmenšiu veľkosť z testovaných dokumentov – 269MB, pričom obsahuje 5 907 961 uzlov.

5.2 Experimentálna zostava

Testovanie prebiehalo na počítači ASUS K55V so 64 bitovým procesorom Intel ® Core ™ i3-3110M CPU @ 2.4GHz, 2.4GHz; 4GB RAM, Windows 8.1. Celá aplikácia, vrátane testovaného holistického algoritmu je implementovaná v jazyku C/C++. Výsledky prezentované v nasledujúcej časti práce sú založené na opakovanom meraní výkonu aplikácie. Ide o vypočítané priemerné hodnoty získané z niekoľkých behov aplikácie s rovnakými vstupnými dátami a rovnakým nastavením. Najhoršie a najlepšie hodnoty sa nezapočítali do výslednej priemernej hodnoty.

5.3 Popis testovania

Na vstupe testovania je vždy vybraný testovaný XML dokument, XPath dotaz a vybraná forma použitej filtrácie – prefixová, postfixová alebo žiadna. Pre každý XML dokument sme otestovali 4 dotazy, spolu ich teda bolo 12. Zároveň platí, že pre každý dotaz bola otestovaná prefixová filtrácia, postfixová filtrácia, a aj spracovanie dotazu bez použitia filtrácie, aby ich bolo možné vo výsledku porovnať.

Pri testovaní sa skúma na koľko je vybraná filtrácia efektívna. V tabuľkách 5.1,5.2, 5.3 máme zobrazené veľkosti streamov jednotlivých dokumentov pri použití *tag streaming scheme* pre testované uzly vetveného dotazu. Pre každý uzol vetveného dotazu je vo výslednej tabuľke uvedený stĺpec *Počet uzlov v streame,* ktorý predstavuje celkový počet uzlov XML dokumentu, ktorý bol odovzdaný algoritmu GTPStack pre daný uzol vetveného dotazu.

[11] http://dumps.wikimedia.org/enwiki/20130805/
[12] http://www.cs.washington.edu/research/xmldatasets/www/repository.html

V stĺpci *Rozdiel* je potom číslo, ktoré predstavuje redukciu oproti použitiu tag streaming scheme.

Hlavnou myšlienkou je sekvenčne spracovávať XML dokument na vstupe a priebežne posielať vyfiltrované uzly algoritmu GTPStack aby vykonával dotaz. Algoritmus GTPStack si na začiatku spracovania XML dokumentu inicializuje potrebné dátové štruktúry. V priebehu spracovávania XML dokumentu vybraná filtrácia plní streamy vetveného dotazu vyfiltrovanými uzlami. Na to aby začal pracovať algoritmus GTPStack, musí byť splnených niekoľko podmienok zároveň:

- Aspoň jeden zo streamov vetveného dotazu musí dosiahnuť veľkosť nastavenej prahovej hodnoty – pre vykonané testovanie bola použitá hodnota 1000.
- Ani jeden stream vetveného dotazu nie je prázdny.
- Všetky zásobníky použité na filtrovanie uzlov sú prázdne. To znamená aj to, že všetky značky v streamoch sú ukončené – majú nastavenú svoju *rightPos*.

Koľko krát boli počas spracovania XML dokumentu splnené všetky tieto podmienky zároveň a teda algoritmus GTPStack mohol začať (pokračovať), nám vo výsledkoch udáva hodnota *Prahová hranica*. *Veľkosť výsledku* je hodnota určujúca počet výsledných n-tic, ktoré vypočítal algoritmus GTPStack. Keďže vytvorené filtrácie filtrovali XML uzly korektne, *veľkosť výsledku* je pre všetky testovania (prefixové, postfixové a bez filtrácie) rovnaký. Porovnávali sme aj celkový počet XML uzlov, ktoré boli predané do streamov pri použití *Tag streaming scheme* (v tabuľkách je to stĺpec *Celkový počet v Tag SS)*, resp. pri použití danej filtrácie (v tabuľkách je to stĺpec *Počet uzlov po filtrácii)*.

Pri testovaní sme merali aj čas. Hodnota *GTPStack čas* udáva celkový čas, ktorý zabrala práca algoritmu GTPStack. No a na koniec je vo výsledkoch uvedená ešte aj hodnota *Celkový čas*, ktorá znázorňuje celkový čas testovania, teda od začiatku spracovania dokumentu až po koniec jeho spracovania.

Pre účely lepšieho porovnania časov bolo pre každý dotaz otestované aj spracovanie dokumentu bez filtrácie. Pri tomto testovaní teda streamy používali *tag streaming scheme*. Pre spustenie GTPStack v priebehu spracovania bez filtrácie museli platiť rovnaké podmienky ako pri použití jednej z filtrácií. Z toho vyplýva, že aj pri spracovaní bez filtrácie sa používali zásobníky, aby sme vedeli, kedy je možné predať algoritmu kompletné značky (mali nastavené *rightPos*) – značky sa ale nefiltrovali.

Názov XML uzlu	Celkový počet výskytov [10^3]
page	2 546
revision	2 546
contributor	2 546
id	7 594
timestamp	2 546
username	2 502
parentid	1 326
minor	779

Tabuľka 5.1 Súhrn počtu vybraných uzlov pre XML dokument enwiki-latest-stub-meta-current26.xml

Názov XML uzlu	Celkový počet výskytov [10^3]
ProteinEntry	262
reference	314
accinfo	312
xrefs	523
xref	1 493
refinfo	314
authors	314
author	5 668
year	314
accession	635
citation	314

Tabuľka 5.2 Súhrn počtu vybraných uzlov pre XML dokument proteinDb.xml

Názov XML uzlu	Celkový počet výskytov [10^3]
C	871
D	871
F	95
G	95
A	95
X	871
Y	871
E	871

Tabuľka 5.3 Súhrn počtu vybraných uzlov pre XML dokument recursiveDocument.xml

5.4 Výsledky testovania

5.4.1 Testovanie dokumentu enwiki-latest-stub-meta-current26.xml

Dotaz 1:

//page//revision//contributor//id

Veľkosť výsledku [10^3]: 2 502

Názov XML uzlu	Prefix		Postfix	
	Počet uzlov v streame [10^3]	Rozdiel [10^3]	Počet uzlov v streame [10^3]	Rozdiel [10^3]
page	2 546	0	2 502	44
revision	2 546	0	2 546	0
contributor	2 546	0	2 546	0
id	2 502	5 092	2 502	5 092

Použitá Filtrácia	Celkový čas[s]	GTPStack čas[s]	Počet uzlov v Tag SS[10^3]	Počet uzlov po filtrácii[10^3]	Prahová hranica
Bez filtrácie	135,609	2,683	15 232	15 232	7 588
Prefix	135,515	2,493	15 232	10 140	2 547
Postfix	142,284	2,486	15 232	10 096	2 547

Dotaz 2:

//page[/id and //revision/contributor/id]

Veľkosť výsledku [10^3]: 2 502

Názov XML uzlu	Prefix		Postfix	
	Počet uzlov v streame [10^3]	Rozdiel [10^3]	Počet uzlov v streame [10^3]	Rozdiel [10^3]
page	2 546	0	2 502	44
id_1	2 546	5 048	2 502	5 092
revision	2 546	0	2 502	44
contributor	2 546	0	2 502	44
id_2	2 502	5 092	2 502	5 092

Použitá Filtrácia	Celkový čas[s]	GTPStack čas[s]	Počet uzlov v Tag SS[10^3]	Počet uzlov po filtrácii[10^3]	Prahová hranica
Bez filtrácie	142,118	3,929	22 826	22 826	5 046
Prefix	141,714	3,635	22 826	12 686	2 547
Postfix	149,058	3,578	22 826	12 510	2 547

Dotaz 3:

//page[/revision/timestamp and //contributor//username]

Veľkosť výsledku [10^3]: 2 502

Názov XML uzlu	Prefix		Postfix	
	Počet uzlov v streame [10^3]	Rozdiel [10^3]	Počet uzlov v streame [10^3]	Rozdiel [10^3]
page	2 546	0	2 502	44
revision	2 546	0	2 546	0
timestamp	2 546	0	2 546	0
contributor	2 502	0	2 546	0
username	2 502	0	2 502	0

Použitá Filtrácia	Celkový čas[s]	GTPStack čas[s]	Počet uzlov v Tag SS[10^3]	Počet uzlov po filtrácii[10^3]	Prahová hranica
Bez filtrácie	141,818	2,603	12 642	12 642	2 547
Prefix	141,044	2,604	12 642	12 642	2 547
Postfix	147,438	2,600	12 642	12 598	2 547

Dotaz 4:

//revision[//id and /parentid and /minor]

Veľkosť výsledku [10^3]: 608

Názov XML uzlu	Prefix		Postfix	
	Počet uzlov v streame [10^3]	Rozdiel [10^3]	Počet uzlov v streame [10^3]	Rozdiel [10^3]
revision	2 546	0	608	1 938
id	5 048	2 546	5 048	2 546
parentid	1 326	0	1 326	0
minor	779	0	779	0

Použitá Filtrácia	Celkový čas[s]	GTPStack čas[s]	Počet uzlov v Tag SS[10^3]	Počet uzlov po filtrácii[10^3]	Prahová hranica
Bez filtrácie	132,690	1,198	12 245	12 245	7 588
Prefix	132,188	1,095	12 245	9 699	5 046
Postfix	131,792	0,979	12 245	7 761	5 046

Vyhodnotenie testov pre XML dokument enwiki-latest-stub-meta-current26.xml :

Po otestovaní vyššie uvedených štyroch dotazov pre dokument enwiki-latest-stub-meta-current26.xml môžeme skonštatovať, že celkový priemerný čas prefixovej optimalizácie je 137,62 s. Naopak, celkový priemerný čas postfixovej optimalizácie dosiahol hodnoty 142,64 s. Keďže celkový nameraný priemerný čas bez filtrácie je 138,05 s, môžeme skonštatovať, že spracovanie prvého testovaného dokumentu s prefixovou filtráciou prebehlo s najlepším časom. Postfixový celkový priemerný čas je o niečo horší ako je to v prípade prefixových časov. Za zmienku stojí aj to, že takmer v každom testovanom dotaze, mala najlepší GTPStack čas postfixová filtrácia, keďže dokázala vyfiltrovať najviac XML uzlov, to ale nevyvážilo väčšiu náročnosť filtrácie. Iba v dotaze č.3 boli GTPStack časy takmer totožné, čo je spôsobené faktom, že bol vyfiltrovaný iba malý počet XML uzlov.

5.4.2 Testovanie dokumentu proteinDb.xml

Dotaz 5:

//ProteinEntry/reference/accinfo/xrefs/xref

Veľkosť výsledku [10^3]: 1 199

Názov XML uzlu	Prefix		Postfix	
	Počet uzlov v streame [10^3]	Rozdiel [10^3]	Počet uzlov v streame [10^3]	Rozdiel [10^3]
ProteinEntry	262	0	245	17
reference	314	0	314	0
accinfo	312	0	312	0
xrefs	281	242	281	242
xref	1 199	294	1 199	294

Použitá Filtrácia	Celkový čas[s]	GTPStack čas[s]	Počet uzlov v Tag SS[10^3]	Počet uzlov po filtrácii[10^3]	Prahová hranica
Bez filtrácie	69,933	0,949	2 904	2 904	1 198
Prefix	69,902	0,933	2 904	2 368	1 198
Postfix	69,746	0,918	2 904	2 351	1 198

Dotaz 6:

//ProteinEntry//reference//refinfo//authors/author

Veľkosť výsledku [10^3]: 5 668

Názov XML uzlu	Prefix		Postfix	
	Počet uzlov v streame [10^3]	Rozdiel [10^3]	Počet uzlov v streame [10^3]	Rozdiel [10^3]
ProteinEntry	262	0	258	4
reference	314	0	314	0
refinfo	314	0	314	0
authors	314	0	314	0
author	5 668	0	5 668	0

Použitá Filtrácia	Celkový čas[s]	GTPStack čas[s]	Počet uzlov v Tag SS[10^3]	Počet uzlov po filtrácii[10^3]	Prahová hranica
Bez filtrácie	71,208	2,929	6 872	6 872	5 511
Prefix	70,880	2,936	6 872	6 872	5 511
Postfix	72,026	2,932	6 872	6 868	5 511

Dotaz 7:

//refinfo[//authors//author and //year and //xrefs]

Veľkosť výsledku [10^3]: 228

Názov XML uzlu	Prefix		Postfix	
	Počet uzlov v streame [10^3]	Rozdiel [10^3]	Počet uzlov v streame [10^3]	Rozdiel [10^3]
refinfo	314	0	228	86
authors	314	0	314	0
author	5 668	0	5 668	0
year	314	0	314	0
xrefs	233	290	233	290

Použitá Filtrácia	Celkový čas[s]	GTPStack čas[s]	Počet uzlov v Tag SS[10^3]	Počet uzlov po filtrácii[10^3]	Prahová hranica
Bez filtrácie	69,264	0,479	7 133	7 133	5 523
Prefix	68,660	0,451	7 133	6 843	5 474
Postfix	69,555	0,439	7 133	6 757	5 339

Dotaz 8:

//reference[//accession and //author and //refinfo and //citation]

Veľkosť výsledku [10^3]: 299

Názov XML uzlu	Prefix		Postfix	
	Počet uzlov v streame [10^3]	Rozdiel [10^3]	Počet uzlov v streame [10^3]	Rozdiel [10^3]
reference	314	0	299	15
accession	312	323	312	323
author	5 668	0	5 668	0
refinfo	314	0	314	0
citation	314	0	314	0

Použitá Filtrácia	Celkový čas[s]	GTPStack čas[s]	Počet uzlov v Tag SS[10^3]	Počet uzlov po filtrácii[10^3]	Prahová hranica
Bez filtrácie	70,444	1,330	7 245	7 245	5 228
Prefix	69,804	1,320	7 245	6 922	5 522
Postfix	70,124	1,319	7 245	6 907	5 387

Vyhodnotenie testov pre XML dokument proteinDb.xml:

Po vykonaní štyroch dotazov pre druhý testovaný dokument proteinDb.xml vyšiel v prípade prefixu celkový priemerný čas 69,81 s. Pri postfixovom testovaní sme namerali celkový priemerný čas 70,36 s. Pri porovnaní s celkovým priemerným časom spracovania dokumentu bez filtrácie, ktorý činil hodnotu 70,21 s, môžeme pozorovať, že celkové priemerné časy spracovania tohto dokumentu boli veľmi podobné. Iba tesne, ale predsa najrýchlejšie sa dokázalo spracovať dokument s prefixovou filtráciou, nasleduje spracovanie bez filtrácie a najdlhšie trvalo dokument spracovať postfixovej filtrácii. Na rozdiel od testovania prvého dokumentu, sa tento krát hodnoty prefixových a postfixových časov príliš nelíšia. Až na dotaz č.6, kde sa vyfiltrovalo iba minimum XML uzlov, postfixová aj prefixová filtrácia dokázali urýchliť beh algoritmu GTPStack. Na koniec môžeme konštatovať ten istý výsledok, ako to bolo v prípade prvého testovaného dokumentu – postfixová filtrácia dokázala pri každom dotaze vyfiltrovať o niečo viac XML uzlov ako je to v prípade prefixovej, avšak časová náročnosť postfixovej filtrácie je priveľká.

5.4.3 Testovanie dokumentu recursiveDocument.xml

Dotaz 9:

//D[//D//E and //C//C//E and //E//Y]

Veľkosť výsledku [10^3]: 577

Názov XML uzlu	Prefix		Postfix	
	Počet uzlov v streame [10^3]	Rozdiel [10^3]	Počet uzlov v streame [10^3]	Rozdiel [10^3]
D_1	871	0	669	202
D_2	679	192	679	192
E_1	577	294	577	294
C_1	679	192	679	192
C_2	577	294	577	294
E_2	577	294	577	294
E_3	679	192	679	192
Y	679	192	679	192

Použitá Filtrácia	Celkový čas[s]	GTPStack čas[s]	Počet uzlov v Tag SS[10^3]	Počet uzlov po filtrácii[10^3]	Prahová hranica
Bez filtrácie	21,420	0,821	6 968	6 968	867
Prefix	20,879	0,721	6 968	5 318	867
Postfix	30,437	0,713	6 968	5 116	676

Dotaz 10:

//F/G/C//C//C//D

Veľkosť výsledku [10^3]: 274

Názov XML uzlu	Prefix		Postfix	
	Počet uzlov v streame [10^3]	Rozdiel [10^3]	Počet uzlov v streame [10^3]	Rozdiel [10^3]
F	95	0	31	64
G	95	0	95	0
C_1	95	776	95	776
C_2	305	566	305	566
C_3	274	597	274	597
D	274	597	274	597

Použitá Filtrácia	Celkový čas[s]	GTPStack čas[s]	Počet uzlov v Tag SS[10^3]	Počet uzlov po filtrácii[10^3]	Prahová hranica
Bez filtrácie	18,948	0,518	3 674	3 674	867
Prefix	18,556	0,371	3 674	1 138	304
Postfix	18,685	0,382	3 674	1 074	304

Dotaz 11:

//A[//C//C//C and //X and //Y and //D//D]

Veľkosť výsledku [10^3]: 71

Názov XML uzlu	Prefix		Postfix	
	Počet uzlov v streame [10^3]	Rozdiel [10^3]	Počet uzlov v streame [10^3]	Rozdiel [10^3]
A	95	0	71	24
C_1	871	0	871	0
C_2	679	192	679	192
C_3	577	294	577	294
X	871	0	871	0
Y	871	0	871	0
D_1	871	0	871	0
D_2	679	192	679	192

Použitá Filtrácia	Celkový čas[s]	GTPStack čas[s]	Počet uzlov v Tag SS[10^3]	Počet uzlov po filtrácii[10^3]	Prahová hranica
Bez filtrácie	21,304	0,726	6 192	6 192	862
Prefix	21,026	0,661	6 192	5 514	862
Postfix	21,208	0,654	6 192	5 490	862

Dotaz 12:

//C[//X and //C and //Y and //E and //D//D]

Veľkosť výsledku [10^3]: 679

Názov XML uzlu	Prefix		Postfix	
	Počet uzlov v streame [10^3]	Rozdiel [10^3]	Počet uzlov v streame [10^3]	Rozdiel [10^3]
C_1	871	0	782	89
X	871	0	871	0
C_2	679	192	679	192
Y	871	0	871	0
E	871	0	871	0
D_1	871	0	871	0
D_2	679	192	679	192

Použitá Filtrácia	Celkový čas[s]	GTPStack čas[s]	Počet uzlov v Tag SS[10^3]	Počet uzlov po filtrácii[10^3]	Prahová hranica
Bez filtrácie	21,374	1,273	6 097	6 097	867
Prefix	21,079	1,228	6 097	5 713	867
Postfix	22,612	1,225	6 097	5 624	867

Vyhodnotenie testov pre dokument recursiveDocument.xml:

Aj pre posledný z testovaných dokumentov recursiveDocument.xml sme vykonali štyri dotazy. Pre spracovanie dokumentu s prefixovou filtráciou je nameraný celkový priemerný čas 20,39 s. Pri testovaní spracovania dokumentu s postfixovou filtráciou sme namerali celkový priemerný čas 23,24 s. Celkový priemerný čas spracovania posledného testovaného dokumentu bez použitia filtrácie je 20,76 s. Podľa očakávania aj toto testovanie dopadlo podobne, ako to bolo v prípade testovania dvoch predchádzajúcich dokumentov. Najmenší čas na spracovanie dokumentu potrebovala prefixová filtrácia, nasleduje spracovanie bez filtrácie, a na koniec postfixové spracovanie. V každom z vykonaných dotazov pre tento dokument, dokázali postfixová, resp. prefixová filtrácia urýchliť čas spracovania dokumentu algoritmom GTPStack. Aj pri testovaní tohto dokumentu môžeme skonštatovať podobný záver, ako to bolo pri testoch predchádzajúcich dokumentov. Postfixová filtrácia síce dokázala v každom prípade vyfiltrovať viac XML uzlov ako prefixová filtrácia, no nedokázalo to vyvážiť väčšie časové nároky na samotnú filtráciu.

6 Záver

Holistické algoritmy vykonávajúce XPath dotazy sú navrhnuté tak, aby pracovali s vopred spracovaným XML dokumentom vo forme indexu. Úlohou tejto práce bolo otestovať ich funkčnosť v prípade, že XML dokument nie je vopred spracovaný, ale čítame ho na vstupe sekvenčne a riadime sa pri tom danou XML Schema validnou k XML dokumentu. Ďalším cieľom tejto práce bolo vytvorenie mechanizmu, ktorý do streamov holistického algoritmu vyfiltruje prefixovo a postfixovo iba relevantné uzly XML dokumentu s ohľadom na zadaný vetvený dotaz. XML Schema nám dovolilo sledovať pozíciu v XML dokumente, čo umožnilo vylepšiť filtráciu.

Základom bolo vytvoriť XML Schema parser, ktorý mal za úlohu vytvoriť orientovaný graf v pamäti počítača podľa zadanej XML Schema. Jednotlivé vrcholy grafu, ktoré predstavovali XML Schema elementy, bolo potom treba označiť spôsobom, ktorý nám umožňoval orientáciu a posun po jednotlivých XML Schema elementoch v závislosti na aktuálne prichádzajúcej značke od XML parsera tak, aby sme v každom bode spracovania dokumentu vedeli, ktorému XML Schema elementu odpovedá aktuálna značka.

Jednou z hlavných úloh tejto práce bolo zredukovať počet XML uzlov v jednotlivých streamoch holistického algoritmu GTPStack, čo by umožnilo zmenšiť jeho pamäťovú a časovú náročnosť spracovania zadaného dotazu. Tento cieľ sa nám podarilo dosiahnuť vytvorením prefixovej a postfixovej filtrácie uzlov XML dokumentu. Podľa vykonaného testovania môžeme konštatovať, že tie XML uzly, ktoré dokázala vyfiltrovať prefixová filtrácia, dokázala vždy vyfiltrovať aj postfixová filtrácia. Postfixová filtrácia k tomu ešte odfiltrovala aj ďalšie značky relevantné pre koreňový uzol vetveného dotazu, avšak to nedokázalo vyvážiť jej celkovú časovú náročnosť.

Pokiaľ ide o časovú zložitosť celkového spracovania XML dokumentu pre zadaný dotaz, z testov vyplýva, že najefektívnejšie je väčšinou spracovanie XML dokumentu s prefixovou filtráciou. V tesnom závese je spracovanie XML dokumentu bez filtrácie, ktorého hodnoty sa často blížili spracovaniu XML dokumentu s prefixovou filtráciou. Spracovanie XML dokumentu s postfixovou filtráciou dosiahlo najhoršie časové hodnoty, čo je spôsobené faktom, že postfixová filtrácia musela v priebehu vkladania XML uzlov do streamu koreňového uzlu vetveného dotazu neustále kontrolovať, či neobsahuje duplicitné XML uzly.

Použitie spracovania XML dokumentu s postfixovou filtráciou je teda vždy efektívnejšie, pokiaľ ide o pamäťovú náročnosť – vyfiltruje vždy najväčší počet XML uzlov. Je však menej efektívne, čo sa týka časovej náročnosti na samotnú filtráciu XML uzlov a teda aj celkovú časovú náročnosť na spracovanie daného dotazu pre XML dokument, čo sa ukázalo byť jej najväčšie slabé miesto.

Ďalším možným vývojom tejto práce by mohlo byť vytvorenie mechanizmu, ktorý by spracoval do orientovaného grafu zadané DTD schéma. DTD síce disponuje menej funkciami ako XML Schema, ale stále je v súčasnosti častejšie používanou schémou aj vďaka jej širokej podpore od vývojárov XML parserov. Ďalšou možnosťou je otestovanie práce s ostatnými holistickými algoritmami.

7 Literatúra

1. **Consortium, W3C.** XML Tree. *w3schools.com.* [Online] http://www.w3schools.com/xml/xml_tree.asp.

2. **Consortium, W3C.** XML DTD. *w3schools.com.* [Online] http://www.w3schools.com/xml/xml_dtd.asp.

3. **Jervis, Michael.** Programming XSLT, XML DTDs Vs XML Schema. *sitepoint.* [Online] 26. 11 2002. http://www.sitepoint.com/xml-dtds-xml-schema/.

4. **Consortium, W3C.** XML Schema. *w3schools.com.* [Online] http://www.w3schools.com/xml/xml_schema.asp.

5. **Consortium, W3C.** XML and XPath. *w3schools.com.* [Online] http://www.w3schools.com/xml/xml_xpath.asp.

6. *DDE: from dewey to a fully dynamic XML labeling scheme.* **Liang Xu, Tok Wang Ling, Huayu Wu, Zhifeng Bao.** s.l. : ACM Press, 2009. International Conference on Management of Data. s. 719-730. 978-1-60558-551-2.

7. **Lu, Jiaheng.** *An Introduction to XML Query Processing and Keyword Search.* Peking : Springer, 2013. 978-3-642-34555-5.

8. *Holistic Twig Joins: Optimal XML Pattern Matching.* **Nicolas Bruno, Nick Koudas, Divesh Srivastava.** s.l. : ACM Press, 2002. International Conference on Management of Data. s. 310-321. 1-58113-497-5.

9. *Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach.* **Jiaheng Lu, Ting Chen, Tok Wang Ling.** Singapore : ACM Press, 2004. Conference on Information and Knowledge Management. s. 533-542. 1-58113-874-1.

10. *On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques.* **Ting Chen, Jiaheng Lu, Tok Wang Ling.** s.l. : ACM Press, 2005. International Conference on Management of Data. s. 455-466. 1-59593-060-4.

11. *DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases.* **Roy Goldman, Jennifer Widom.** s.l. : Morgan Kaufmann Publishers Inc. San Francisco, 1997. Proceedings of VLDB 1997. s. 436-445. 1-55860-470-7.

12. *Optimal and Efficient Generalized Twig Pattern Processing: A combination of Preorder and Postorder Filterings.* **Radim Bača, Michal Krátky, Tok Wang Ling, Jiageng Lu.** s.l. : Springer-Verlag, 2013, s. 369-393.

13. **Wikipedia.** Wikipedia the Free Encyklopedia. *Tree traversal.* [Online] [Dátum: 4. 4 2014.] http://en.wikipedia.org/wiki/Tree_traversal.

8 Prílohy

CD	**Obsah adresára**
Aplikacia\	Aplikácia použitá v práci
Aplikacia\Readme.txt	Pokyny k spusteniu aplikácie
Test_data\	Testovacie dáta použité v práci